NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN IMPLEMENTATION OF A DATA DEFINITION FACILITY
FOR THE GRAPHICS LANGUAGE FOR DATABASE

by

Michael L. Williamson

December 1988

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified		1b Restrictive Markings		
2a Security Classification Authority		3 Distribution Availability of Report		
2b Declassification/Downgrading Schedule		Approved for public release; distribution is unlimited.		
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)		
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol (If Applicable) 37	7a Name of Monitoring Organization Naval Postgraduate School		
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000		7b Address (city, state, and ZIP code) Monterey, CA 93943-5000		
8a Name of Funding/Sponsoring Organization Naval Data Automation Command	8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number		
8c Address (city, state, and ZIP code) Washington Navy Yard, Washington, DC 20374-1662		10 Source of Funding Numbers		
		Program Element Number	Project No Task No Work Unit Accession No	

11 Title (Include Security Classification) An Implementation of a Data Definition Facility for the Graphics Language for Database

12 Personal Author(s) Williamson, Michael L.

13a Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report (year, month,day) December 1988	15 Page Count 89

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)
Field	Group	Subgroup	Database, Object-oriented, GLAD, DBMS, BASIS

19 Abstract (continue on reverse if necessary and identify by block number

This research is an implementation of the data definition facility for the Graphics Language for Database (GLAD). GLAD is a graphics-oriented database management system which is primarily concerned with ease of learning and efficiency of use. The system uses an object-relationship approach to database design. Entities of the database are represented graphically as objects. With this method, users can visualize the schema of the database and can quickly comprehend how the entities relate. Every effort has been made to design GLAD so that a new user can quickly learn to create and manipulate a database without the need of a dedicated database administrator.

20 Distribution/Availability of Abstract	21 Abstract Security Classification	
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	Unclassified	
22a Name of Responsible Individual C. T. Wu	22b Telephone (Include Area code) (408) 646-3391	22c Office SymbolT. 52Wq

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   security classification of this page

Approved for public release; distribution is unlimited

An Implementation of a Data Definition Facility for the Graphics Language for
Database

by

Michael L. Williamson
Lieutenant, United States Navy
B.S., United States Naval Academy, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

ABSTRACT

This research is an implementation of the data definition facility for the Graphics language for Database (GLAD). GLAD is a graphics-oriented database management system which is primarily concerned with ease of learning and efficiency of use. The system uses an object-relationship approach to database design. Entities of the database are represented graphically as objects. With this method, users can visualize the schema of the database and can quickly comprehend how the entities relate. Every effort has been made to design GLAD so that a new user can quickly learn to create and manipulate a database without the need of a dedicated database administrator.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The United States Federal Government is the largest single user of computers in the world. In 1986, its investment in automated data processing (ADP) equipment and services amounted to more than 15 billion dollars. This budget included ADP for defense and national security, education, national energy programs, social welfare, and tax programs. [Ref. 1:p. 6] Of the different uses of computers, one of the most important is database management.

The handling of large quantities of information requires a Database Management System (DBMS) with a definition facility for creating the database schema and a manipulation facility for reviewing and updating the information [Ref. 2:p. 1]. In addition, it must be efficient and convenient to use. Modern DBMS are producing efficient results but most require a dedicated database administrator to handle definition of the database schema and the complex transactions which are common in today's world. While this is not a problem at large computer centers, the increasing use and sophistication of the microcomputer for databases make the need for an easy to use system all the more important.

For these reasons, the Graphics Language for Database (GLAD) is being developed. It is built on the principle that a DBMS must provide a good user interface to make the system easy to learn and use for both experienced and novice users. The best approach appears to be a visual representation of the database schema. A brief description of GLAD is reproduced here.

> GLAD utilizes a bit-mapped, high-resolution graphics display terminal. The screen consists of two types of windows: schema and operation window. In the schema window, GLAD provides an elegant visual representation of real world abstraction concepts most semantic data models support: aggregation,

1

generalization, classification, and association. In the operation window, GLAD describes objects, displays results, and allows users to specify queries. Windows can be opened, closed, scaled, and moved at the user's will. [Ref. 3:p. 4]

A more complete description is contained in Naval Postgraduate School Report NPS52-87-030, GLAD: *Graphics Language for Database.*

This thesis discusses the design issues and the implementation of the data definition facility for GLAD. Its purpose is to show that a user interface which can be used by novice database users is possible for even the most complex database. To demonstrate this point, a sample schema is constructed in Appendix A.

Chapter II provides background on Microsoft Windows operating environment and ACTOR object-oriented programming language which are used to implement GLAD. This includes a discussion of the human factors in the design and development of the system. Chapter III discusses object-relationship data models and how they are implemented in GLAD. The implementation of the data definition facility, referred to as the Data Definition Language (DDL) is discussed in Chapter IV. This discussion includes problems encountered and their solutions, and benefits of using an object-oriented language for the development of the system. Chapter V contains the conclusions of the research effort involved in the development of the system. Appendix A shows the definition of the database specified by the Bases and Stations Information System (BASIS) Administration System, Navy Regional Data Automation Center (NARDAC) San Diego, Project Number X1AD001. Appendix B is a listing of the program for the data definition facility.

II. BACKGROUND

A. GLAD REQUIREMENTS

GLAD is designed for use on a microcomputer running the DOS operating system and can only be used in the environment provided by Microsoft Windows. Windows requires a minimum of 512K bytes of RAM memory but recommends 640K bytes for multiple applications. [Ref. 4:p. xii] Although Windows does not require it, GLAD requires the use of a mouse to receive inputs from the user (future versions will be written for use without a mouse). GLAD is written in the Actor programming language, but once compiled, can be run without Actor as a stand alone application. GLAD has been tested on a variety of computers including the Zenith Z-248, which is now being used at most U. S. Navy installations. All hardware required to run GLAD is standard on the Z-248 except the mouse. This means the GLAD system can be used at all installation with a minimum of modification and cost.

In addition, the ease of use of the system means that little training will be required for users to begin using the system effectively. This chapter will describe the research behind the idea of graphics oriented interfaces andshow the benefits of using Microsoft Windows and Actor which make GLAD an easy to learn and use database system.

B. HISTORICAL RESEARCH

1. Graphics Oriented Computer Interfaces

The ideas behind graphics oriented computer interfaces first originated in the early 1960's with a graduate student at the University of Utah, Alan Kay. He

was convinced at that time that the changes in technology would eventually make it possible to put a room-sized, million-dollar computer into a package the size of a notebook. With these technological advances, he felt it would be possible for every person to own a personal computer. Kay proposed to Xerox Corporation his idea of apersonal computer which he called the Dynabook. In 1971 the Xerox Palo Alto Research Center began a project to develop the Dynabook. In 1973, a desk-sized Dynabook became available for research.

The Dynabook used a system of windows to display different kinds of information including output from programs, mail, documents, menus, diagrams, and status information. Users focus their attention on a window by pointing at it with their finger (the mouse was introduced at a later time). This allows users to work on many different things at once since they can suspend their activity in one window and resume a different activity in another window simply by moving the pointing device. Initial tests were conducted using 250 children (6-15 years old) and 50 adults. The tests showed that non-specialists can learn and use the computer effectively when given a highly interactive, visual environment. [Ref. 5:pp. 442-443]

Other research in graphics oriented interfaces being conducted at about the same time led to a term "Direct Manipulation," originally coined by B. Shneiderman in 1974. According to his definition, a direct manipulation interface must display the following properties:

- Continuous representation of the object of interest.
- Physical actions or labeled button presses instead of complex syntax.
- Rapid incremental reversible operations whose impact on the object of interest is immediately visible. [Ref. 6:p. 91]

These properties give a direct manipulation screen a great deal of power. The continuous representation of an object of interest ensures the user is always aware of what is happening. The use of physical actions instead of complex syntax gives the user a feeling of control and makes the system much easier to learn and to use. Both of these ideas were already implemented by Xerox on the Dynabook. Shneiderman's research simply gave these ideas a more theoretical foundation on which to stand. The third item, rapid operations whose impact is immediately visible, was not possible at this early date. It was not until faster processors were available that the true value of this property was realized.

Shneiderman suggested that a system with these properties would have the following virtues:

- Novices can learn basic functionality quickly, usually through a demonstration by a more experienced user.
- Experts can work extremely rapid to carry out a wide range of tasks.
- Knowledgeable intermittent users can retain operational concepts.
- Error messages are rarely needed.
- Users can see immediately if their actions are furthering their goals, and if not, they can simply change the direction of their activity.
- Users have reduced anxiety because the system is comprehensible and because actions are so easily reversible. [Ref. 6:p. 91]

Although the tests using the Xerox Dynabook proved the validity of these virtues, the Apple Macintosh was the first successful commercial computer to prove that these concepts were supportable in the real world environment of computer science. Microsoft has taken these ideas and developed an environment based on this research which displays many of the benefits discussed. These will be discussed in Section B of this chapter.

2. Human-Computer Interaction

Ricky Savage and James Habinek conducted a study on Human-Computer Interaction in 1984. Their purpose was to

> ...design and evaluate a user interface that would satisfy a broad spectrum of users, from competent system programmers to complete novices. The goal was to produce an interface that would easily guide the first-time user through a series of menus to the desired procedure, but at the same time provide the experienced user with the flexibility to take time and effort-saving short-cuts. [Ref. 7:p. 166]

Their primary concern was to provide easy access to the entire system without constant reference to manuals. To do this they designed a system of multi-level, hierarchical menus. They determined that this configuration gave the novice user the necessary step-by-step interface to lead them to the required system functions. For experienced users, abbreviated forms or shortcut commands were needed so that skilled users were not penalized by the slower manipulation of the hierarchy of menus.

The results of this research suggest that the hierarchical design of the interface is compatible with user expectations of the system. The fact that humans are accustomed to the presentation of information in a hierarchical form makes it easier for them to understand and form a cognitive model of the system in their minds. This is especially true when consistency is designed into the various parts of the menu system. A second conclusion indicated that users preferred shorter menus with more levels rather than fewer, long menus. These results suggest that it is easier to follow a hierarchical path than to search through long menus for the proper function. [Ref. 7:pp. 184-185]

C. MICROSOFT WINDOWS

1. Advantages of Windows

Microsoft Windows was chosen as the environment inwhich to implement GLAD because it effectively uses the research techniques discussed to provide a user friendly, graphical-based windowing environment. In Windows, all programs run in a window. Although a single program can create many windows, usually the program has a single main, or top-level window. Since all windows look and work the same regardless of the application, the appearance and interface with the user is the same. This results in the same type of easy to use interface which was first researched using the Dynabook computer. In addition, Windows also provides a multitasking capability which allows more than one program to be running at the same time. In a database system, this is especially desirable since it allows the user to work with more than one database or be looking at two different portions of the same database concurrently. A sample Windows screen running multiple applications is shown in Figure 1.

Each program is identified by a caption bar at the top of its main window, and most functions are initiated through the program's menus located just below the caption bar. This gives Windows a consistent appearance and command structure across many different applications which make the system easier to learn and to use. A special type of window, called a dialog box, is generated by a main or active window and can never stand alone. It is used to request data and command inputs, or to issue warnings or error messages to the user. Basically, a dialog box is used whenever communication between the user and the computer is necessary to continue the program.
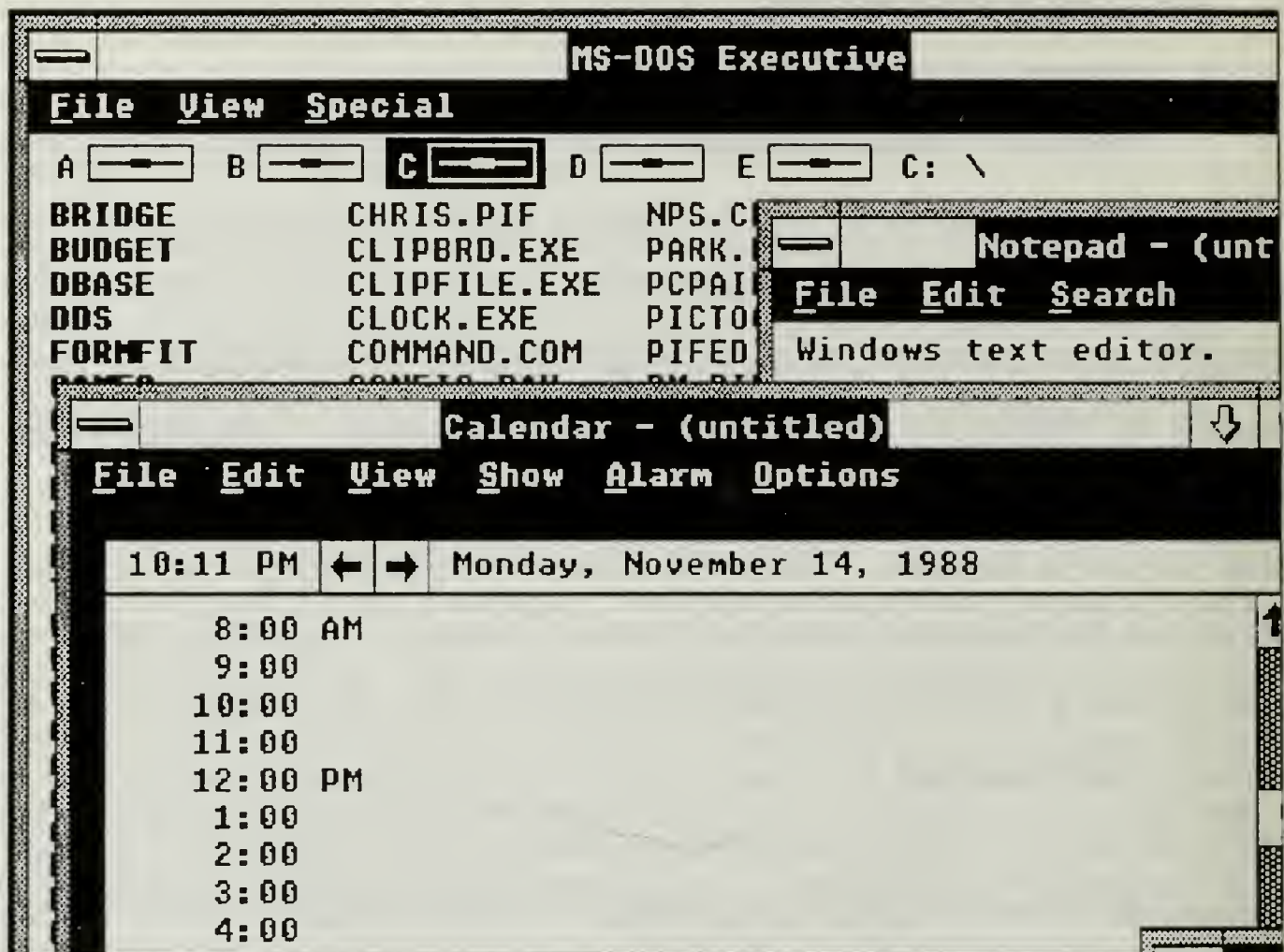
Figure 1. Sample Microsoft Windows Screen

2. Objects and Messages

Everything in Windows is an object; a group of items that when associated together can be referred to as a whole. The object then is a cohesive unit with a unique name, called a handle. An application window is an object used to display information to the user. The parts of the window include the menu items, the scroll bars, and any item related to controlling the window. The window itself can be treated as a whole by calling its handle or an individual part of the window can be manipulated by calling its handle. Handles in Windows are all 16-bit names which are unique to the particular object.

Each time an application is started, an instance of that application is created. Since Windows is a multi-tasking environment, there can be more than one instance of an individual application running at any one time. Each instance of the application is referenced by a different handle.

The only method Windows has of sending or receiving information to or from an application is through the use of messages and events. The information transmitted in a message includes the handle of the target window, the message number, the parameters, the time at which the message was sent, and the mouse position. An event is anything that happens that affects something else. An event can be caused by such things as moving the mouse or pressing a mouse button. MS-Windows then sees this event and sends a message to the application which controls the window in which the mouse was clicked. For each action or event that affects a window, there is an appropriately named message to inform the application. As an example, WM-LBUTTONDOWN is a message that would be sent by Windows to the handle of an application to signal that the left mouse button has been pressed while the cursor was in its window. Messages can also be generated by the clock, keyboard, or by other tasks running within Windows. This procedure can be summarized by the statement, "an event is tied to a message, a message is tied to a handle, and a handle is tied to an object." [Ref. 8: p.71]

Windows performs message-routing and scheduling operations in a nonpreemptive fashion. All applications must abide by the message-passing rules so that no process gets an unfair share of processor cycles. Once an application receives a message, however, it maintains control of the CPU until the message has been processed. Windows can be thought of as a large switchboard, routing messages to their destinations and queuing them until they can be processed.

D. ACTOR PROGRAMMING LANGUAGE

The C language has traditionally been used for programming windows. In 1987, a new language called Actor was introduced. It is based on the principles of object-oriented programming first introduced in the early 1970's in the language, Smalltalk. Object-oriented languages are radically different from procedural languages with which most software developers and microcomputer users are familiar. Programming is accomplished solely by sending messages to objects. [Ref. 9:p. 148] While this design satisfies both the regularity and the simplicity principles, [Ref. 5:p. 464] it tends to unsettle many new users and has contributed to the slow acceptance of object-oriented programming. Actor offers some help not available from other object-oriented languages in that its syntax is similar to Pascal and C. This does not make object-oriented programming easier to learn, but it helps to ease the transition.

Another aspect of Actor which helps to ease the transition is the close interaction between the language and the Microsoft Windows environment. Actor is a Windows application and can only be used in that environment. Actor provides a full set of interface functions to Microsoft Windows. Windows, menus, dialog boxes, accelerator keys, and icons can be defined for use in application programs.

Like everything in Actor each window is an object and commands are sent in the same way that all objects receive commands; through message sent to it. Clearly an understanding of Windows aids in understanding Actor, and vice versa. A more complete description of object-oriented programming principles is presented here to show how these features are exploited in GLAD.

1. Object-Oriented Programming

All object-oriented languages employ a data or object-centered approach to programming. Instead of passing data to procedures, you ask objects to perform operations on themselves. Objects can be numbers, strings, graphics, and anything else which can be represented in the computer. These elements are not passive like data in procedural programming languages and each has characteristics of both data and programs. Each object has a corresponding list of functions, called methods, which can be performed on the object. When a message is received by an object, it checks its list of methods for a match. If one is found, then that method is carried out on the object. [Ref. 9:p. 148] So each object is similar to a module in procedural programming languages and each method is similar to a procedure or function. The programmer only needs to know what objects are available and their corresponding methods. He does not need to know the details of how the method is implemented.

In general, each object acts as an autonomous entity that is only responsible for its own behavior. All of the information relevant to the individual behavior of an object is contained in that object. This might appear to be a waste of memory because if each object had to contain its own **print** method, for example, there would be a great deal of redundancy. Consider each integer containing its own copy of the method **print**. Object-oriented programming languages handle this problem with the use of classes and inheritance between classes.

The class provides all the information necessary to construct and use objects of a particular kind, called its instances. Each instance has one class while a class may have multiple instances. Each class also provides storage for methods. The methods reside in the class since all instances of a class have an identical set of

methods. Methods may allocate temporary variables for use during the execution of the method. These temporary variables are like the local variables in a Pascal procedure.

Each class can be stored in a library for use with many different applications. Actor comes with more than 90 predefined object classes and hundreds of methods. [Ref. 8:p. 407-534] The classes which are used to implement the Data Definition Language of GLAD are included in Appendix B.

2. Inheritance

Inheritance enables programmers to create new classes of objects by specifying the differences between a new class and an existing class instead of starting from scratch each time. This is exactly what Actor has done with the Int and Real classes. By grouping them under a superclass called Number, many of the methods which are used by both types of data can be written only once and these methods are then inherited by the subclasses. This forms a hierarchical tree type structure where the root of the tree is a class called 'object'. [Ref. 5:p. 455] A portion of the Actor tree structure is shown in Figure 2. Note that inheritance in Actor is not limited to one level. When a message is sent to an object, if there is no method in that object which matches the message, the superclass of the object is checked for a match. This search process continues until the Object class at the root is reached. If a matching method is not found, an error message is sent. So inheritance allows a large amount of code to be reused, thus saving precious memory space and shortening development time.
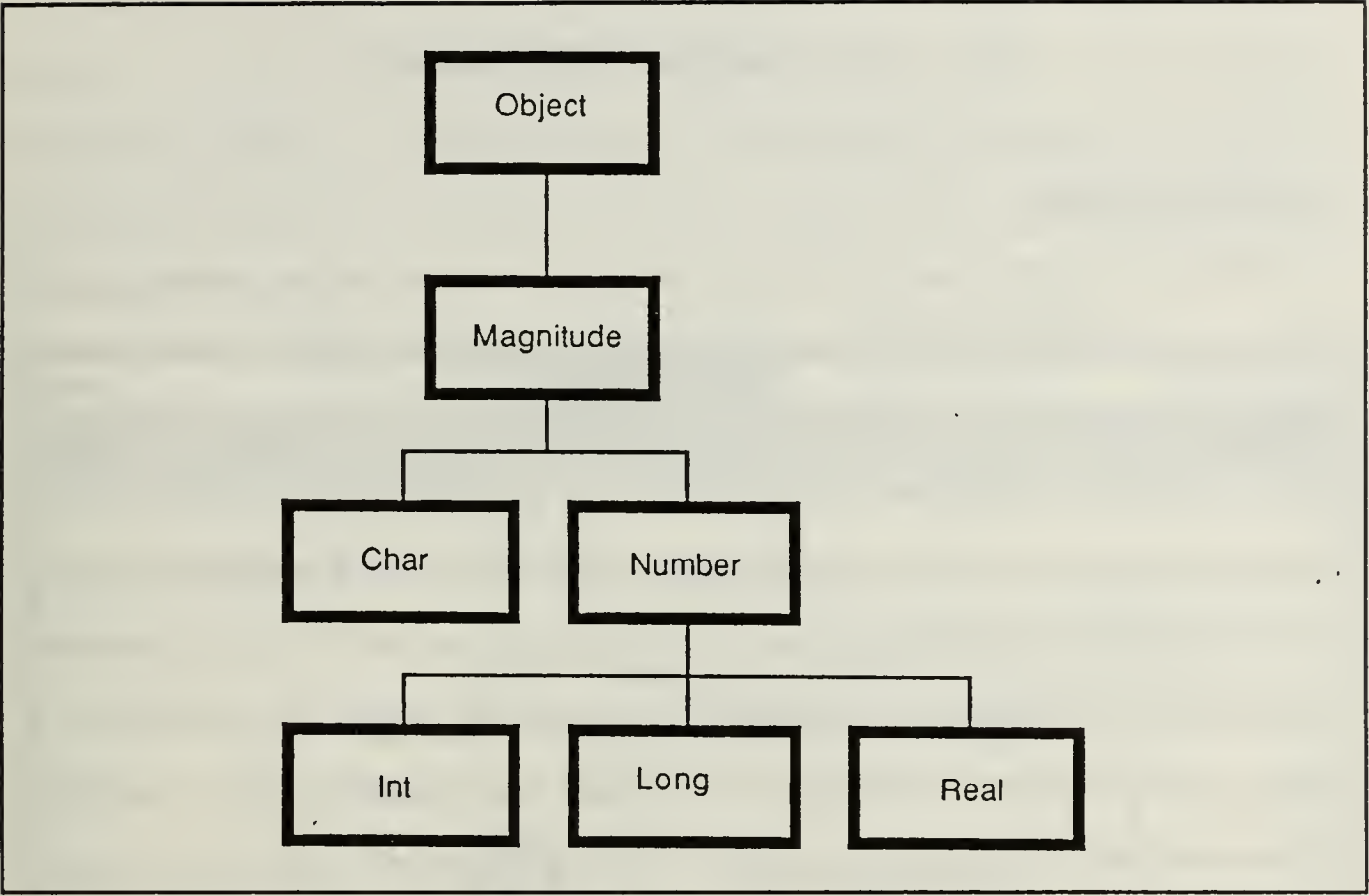
Figure 2. Actor Tree Structure

III. GLAD DATABASE MODEL

A. USING GLAD

The GLAD system consists of a top level window which allows access to other windows and controls functions that relate to the database as a whole. These include creating, modifying, opening, and removing databases. The top-level window is shown in Figure 3. Creation or modification of a database schema are managed through the use of a data definition language. In general, only a dedicated database administrator (DBA) will have access to this facility, as well as, to the database removal facility. Opening of a database for regular use should be available to all persons who have access to the system. Although access controls are not currently implemented, they can easily be handled in GLAD by requiring the user to enter a user code and password in order to use the DDL or the Remove function.

The top level window and the data manipulation language of GLAD have been developed by previous thesis students and by Prof. C. Thomas Wu of the Naval Postgraduate School. The next section will give a brief introduction into how the database system is organized and how it was implemented.

B. DATA MODEL

A data model is "a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints." Most databases currently in use are based on one of the classic data models; they are either relational, E-R, semantic, or infological. The GLAD interface is based on a concept called object-relationship model. In this model, the user is given a visual representation of the collection of basic objects called entities, and the relationships

14

among them. An entity is an "object that exists and is distinguishable from other objects. A relationship is an association among several entities." The distinction between entities is accomplished by associating attributes with each entity.[Ref. 2:p. 6] An entity in GLAD is represented by its name inside a rectangular box.
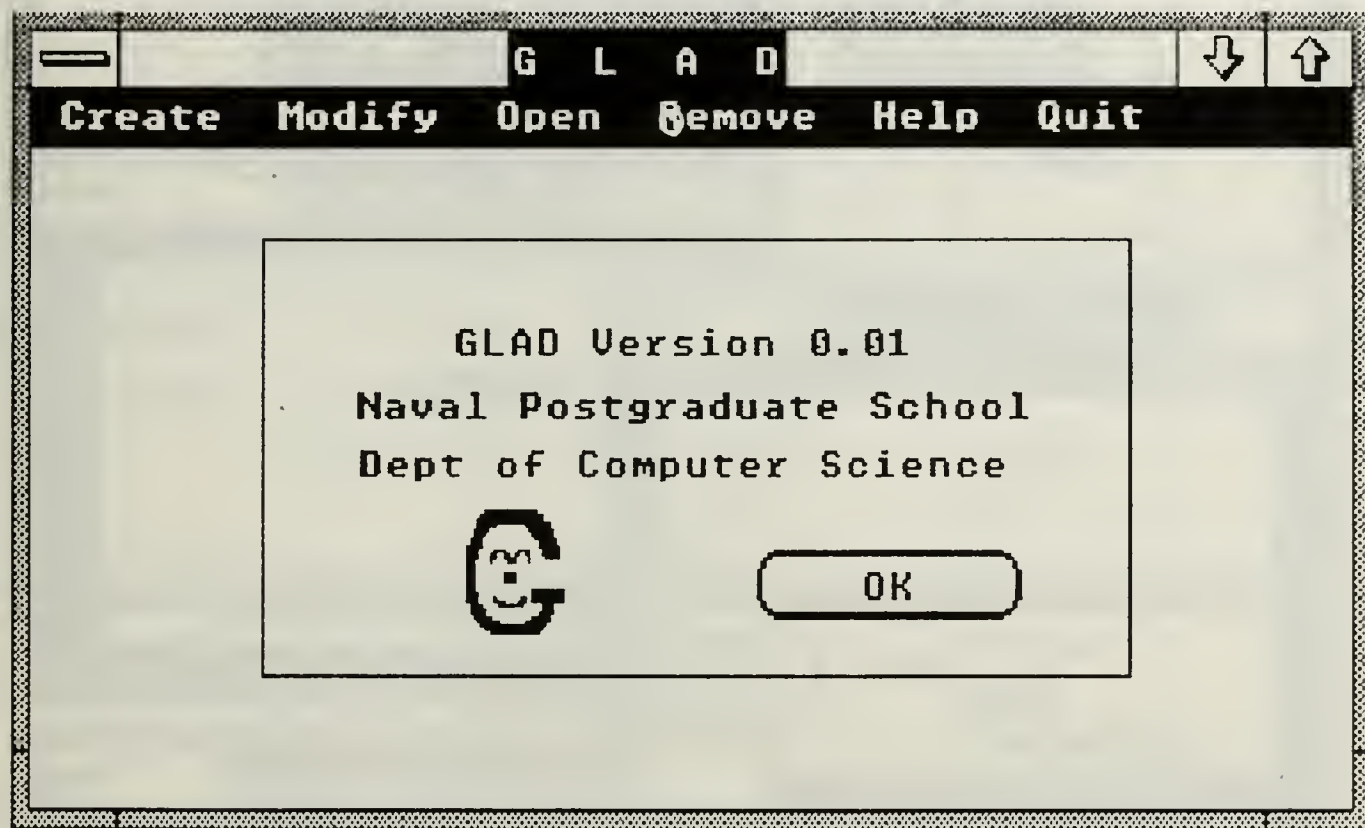
Figure 3. GLAD Top-Level Window

Figure 4 shows a sample database for a fictitious university. In it, the objects DEPT, EMPLOYEE, and EQUIPMENT are entities. Figure 5 shows the attributes associated with each entity. The attribute list includes its the name of the attribute and the data type which it represents. Although not shown, each of the objects is of a different color when displayed on a color monitor. Each attribute window has a border of the same color as the object which it describes.
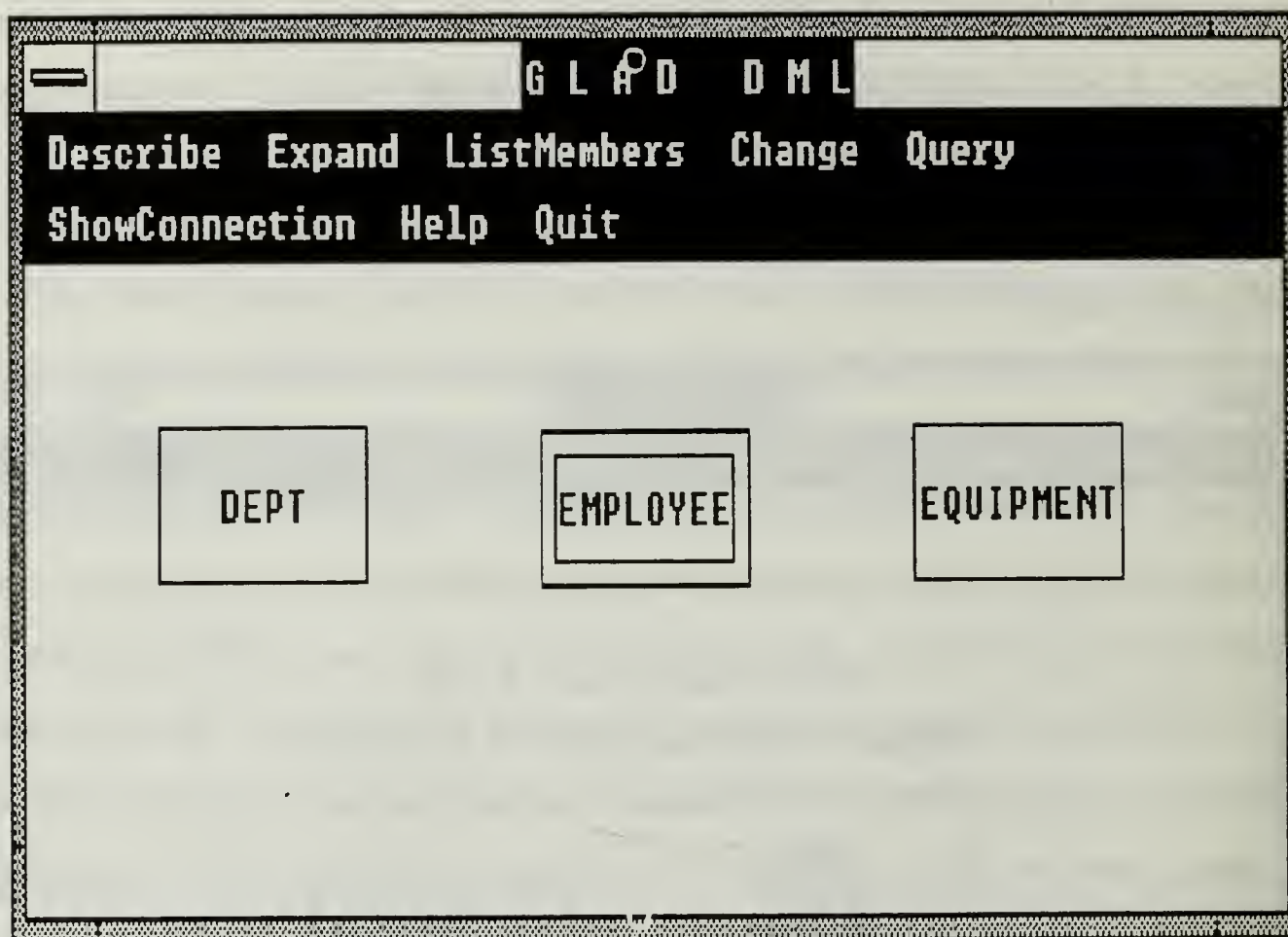
Figure 4. University Database

1. Aggregation

The attributes can be of a system-defined type or a user-defined type. At this stage in the development of GLAD, system-defined types are atomic; that is, they consist of exactly one system defined base object (i.e., integer, string, real, date, money, or boolean). Future versions will include set types for recursive objects. User-defined types refer to other entities within the database. They can be atomic or non-atomic. Non-atomic objects are a collection or aggregation of more than one sub-object (i.e., WorksFor and Belongs are of type DEPT, which is an aggregation of Name and Chair). User-defined types will be shown in reverse video with background color the same as the color of the entity object it represents.

16
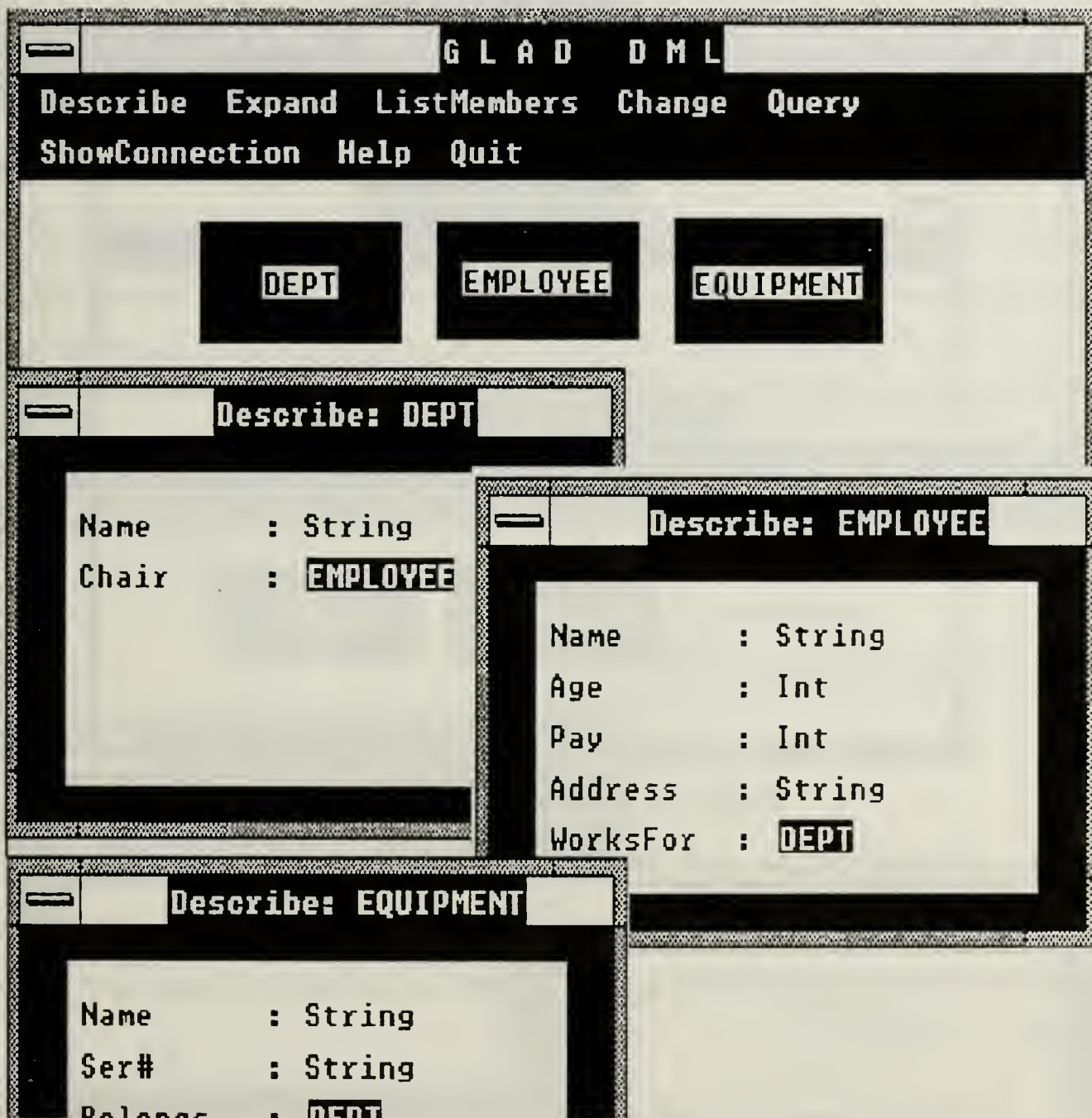
Figure 5. Attributes for University Entities

Figure 6 shows the relationship between the objects of the database. Note that since EMPLOYEE and EQUIPMENT contain attributes of type DEPT, they are shown as being related to DEPT.

17
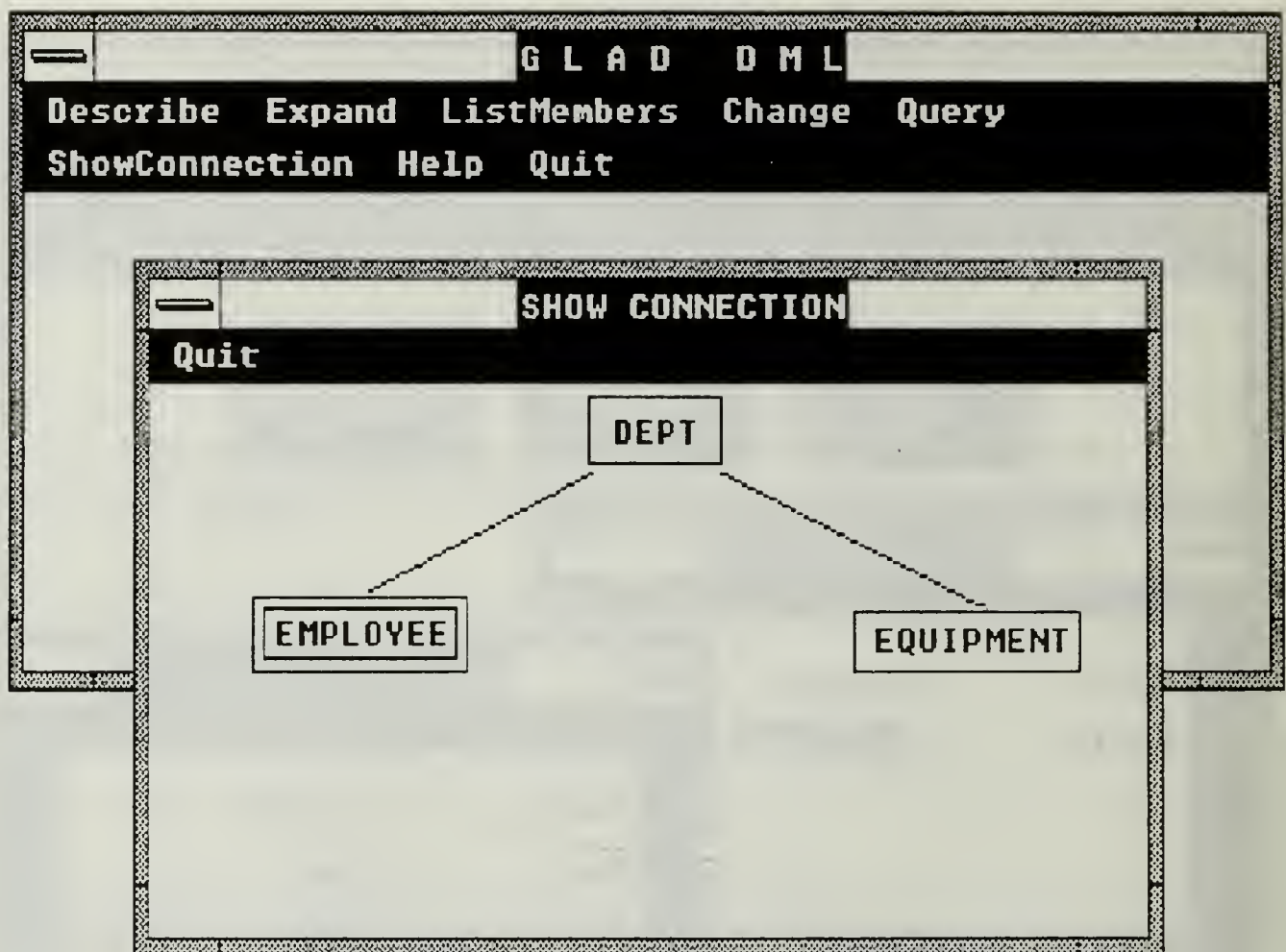
Figure 6. Entity Relationships

2. Generalization and Specialization

Two other relationship types which can be represented in GLAD are generalization and specialization. These terms are defined in *Database Systems Concepts* as follows:

- Generalization is the result of taking the union of two or more (lower level) entity sets to produce a higher-level entity set.

- Specialization is the result of taking a subset of a higher-level entity set to form a lower-level entity set. [Ref. 2:pp. 37-38]

These are easily represented in the object-relationship model through the use of nested objects. Figure 7 shows an expansion of the EMPLOYEE object. From this figure, it is easy to visualize that EMPLOYEE is actually made up of its

18

attributes and two other non-atomic entities, FACULTY and STAFF. So EMPLOYEE is a generalization of FACULTY and STAFF, while FACULTY and STAFF are specializations of EMPLOYEE. Referring again to Figure 4, note that the EMPLOYEE object has a double rectangle. This is how GLAD represents an object which is a generalization of lower-level objects. Specialization can continue to any number of levels. In this case, STAFF is in turn a generalization of TYPIST and TECH as shown in Figure 8.
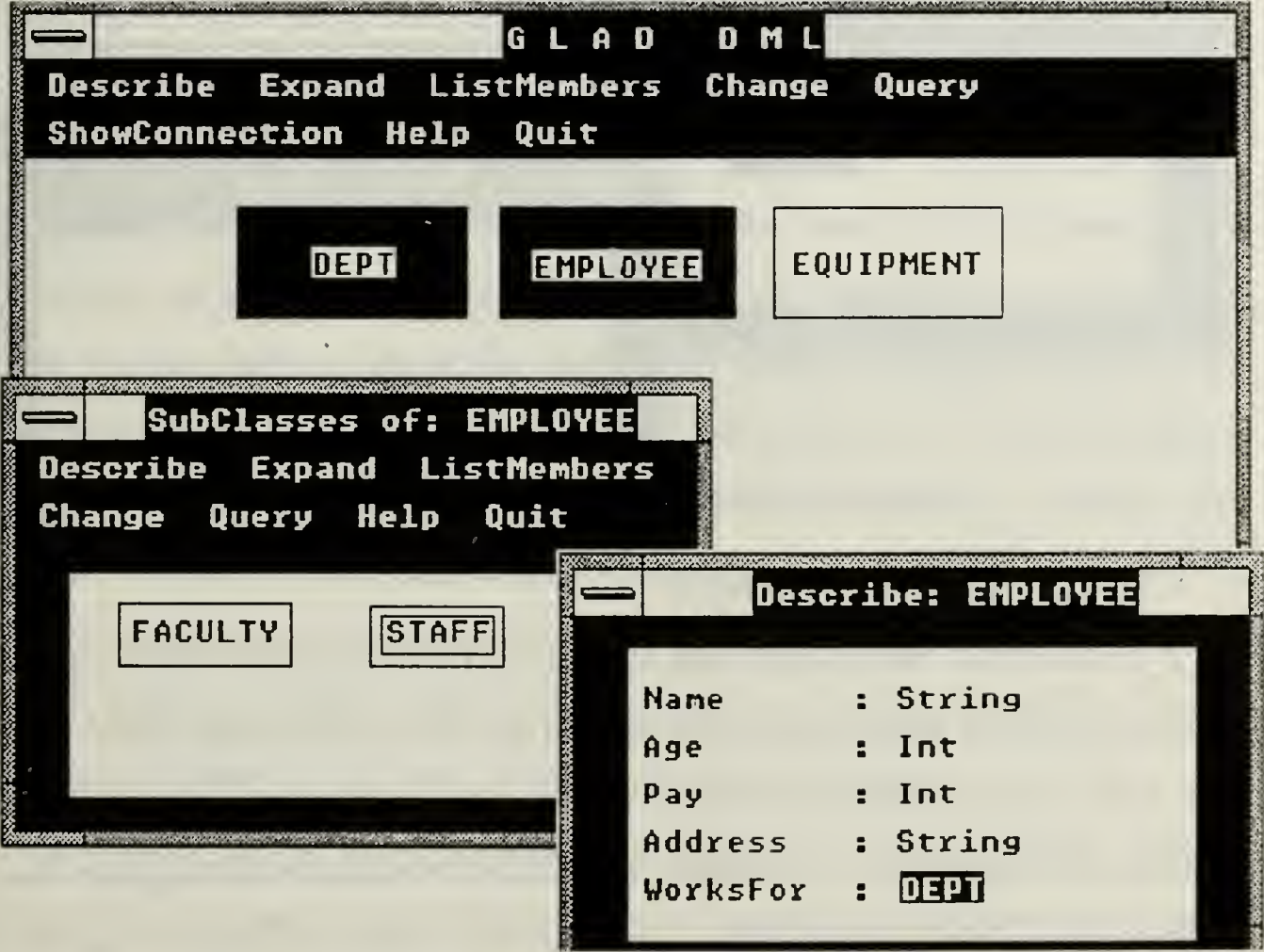
Figure 7. Expansion of EMPLOYEE Object

This section was intended to give the reader an idea of the power and elegance of the GLAD Data Model. The next section will discuss the implementation of the GLAD system in more detail.
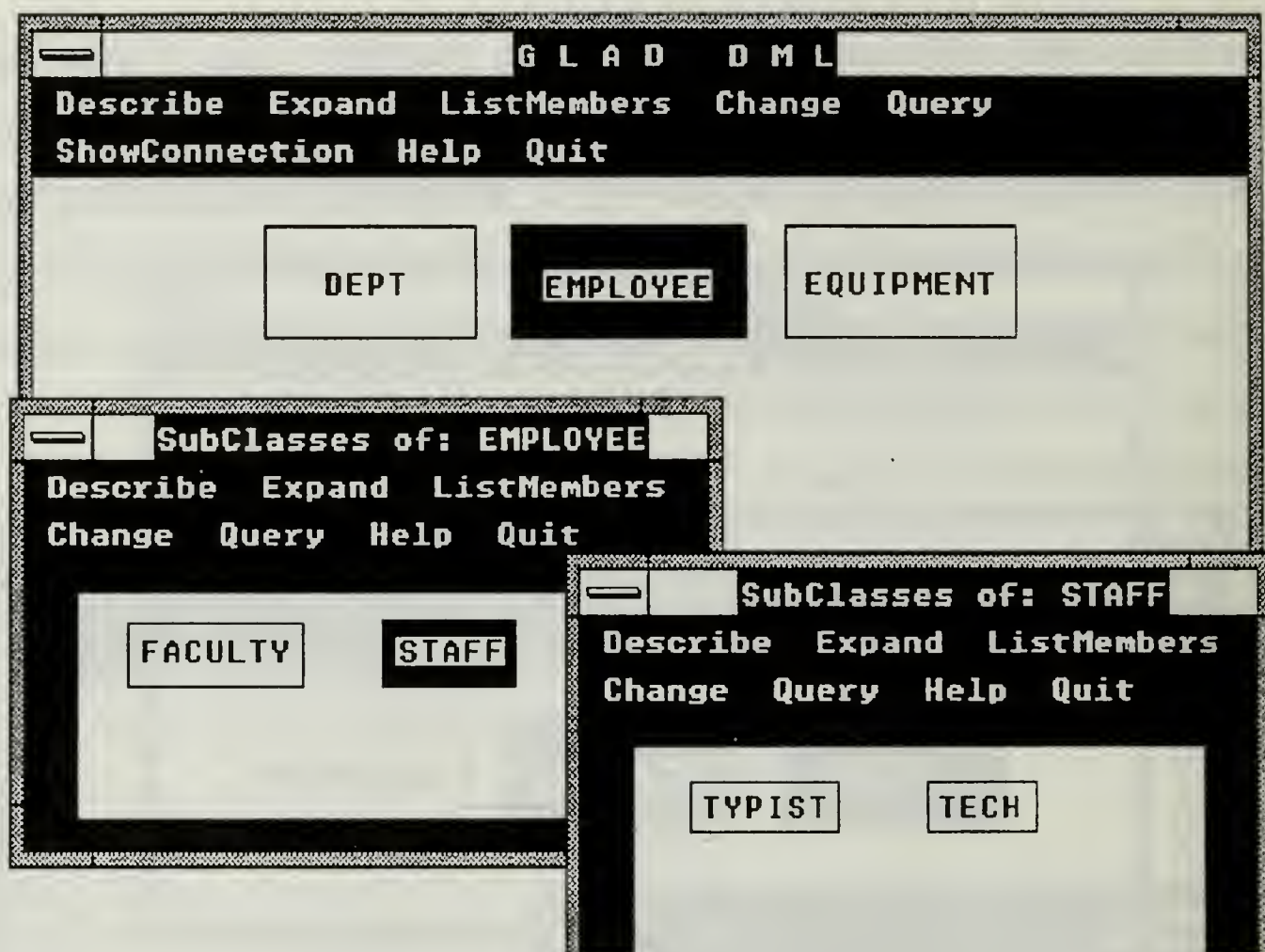
Figure 8. Expansion of STAFF Object

C. USING THE DML WINDOW

It is clear that the graphic-interface offered by Microsoft Windows is an excellent means of displaying the attributes of the object-relationship model. Since each entity is represented by an object, Windows object centered design is ideal for handling interaction with the database, and Actors object-oriented language means programming of the interface can be easily conceptualized. In addition, the ability to call different windows, overlay them, and destroy them offers a great deal of flexibility over which the user has complete control.

Figure 4 showed a sample database schema as displayed in the data manipulation window of GLAD. The window is identified by the title "GLAD

DML" where DML is Data Manipulation Language. This is the primary window interface for database users.

As discussed in Chapter II, the menu is the means by which the user interfaces with the system. The research indicated that short, hierarchical menus are better for the novice user but that short cut methods should be available for more experienced users. In GLAD, the menus have been limited to a maximum of eight menu items. In addition they are, for the most part, only one level. By keeping the menus short, novice users can quickly learn and manipulate the available functions. By using only one level for most functions, short-cuts for experienced users are not necessary.

In order to view the attributes of any object, the arrow, which is controlled by the mouse, can be moved to any object and the left mouse button clicked. This results in the selection of the object, indicated by a green background color and a wider than normal rectangular border on the selected object. Once the object is selected, the user simply selects Describe from the DML menu. The describe window with the list of attributes is automatically displayed.

The Expand menu item is used to display the nested objects within a generalized entity. Selecting EMPLOYEE and then selecting Expand will result in the subclasses of EMPLOYEE being displayed as shown in Figure 7.

Of course, the primary purpose of a database is for the user to retrieve information from the system. GLAD offers two methods of retrieving and viewing data. ListMembers allows the user to view the database either one entry at a time or all at once. These options are shown in Figure 9. Query will offer a full range of query functions which will allow the user to sort the database on individual field

21

characteristics or combinations of characteristics. Work is still underway in this area to develop a powerful, yet easy to use query language interface.

Figure 9. Views of the Database

Change allows the user to update the data in the database. The user can add, delete or modify any record. ShowConnection gives a visual display of the relations within the database as shown in Figure 6. Help offers on-line help routines and Quit causes the system to exit the DML routine.

22

IV. DATA DEFINITION LANGUAGE

A. BACKGROUND

The data definition language is used to define new or modify existing database schema. Included in this facility are provisions to add new objects, delete objects, define attributes, define specialization of objects and save these in a way that can be used by the system.

DDL is invoked by selecting either the Create or Modify menu item from the top-level GLAD window. If Create is selected, the user is asked for the name of the new database using a dialog box as shown in Figure 10A. If Modify is selected, the dialog box shown in Figure 10B is displayed. This dialog displays all of the database schema currently in the system and allows the user to select the desired schema. Like everything in the Windows environment, more than one instance of any database can be called from the top-level window allowing a user to modify more than one database schema at the same time.

The remainder of this chapter discusses some of the design considerations that went into the development of the DDL facility. Appendix B shows the actual classes and methods that are used. Note that each of the windows and dialog boxes described are controlled by a separate class. For example, the main DDL window is controlled by the DDWindow Class while the AttribDialog Class controls the dialog for entering attributes for each entity. These classes will be discussed along with the interface which the user sees when defining a new or modifying an existing database.
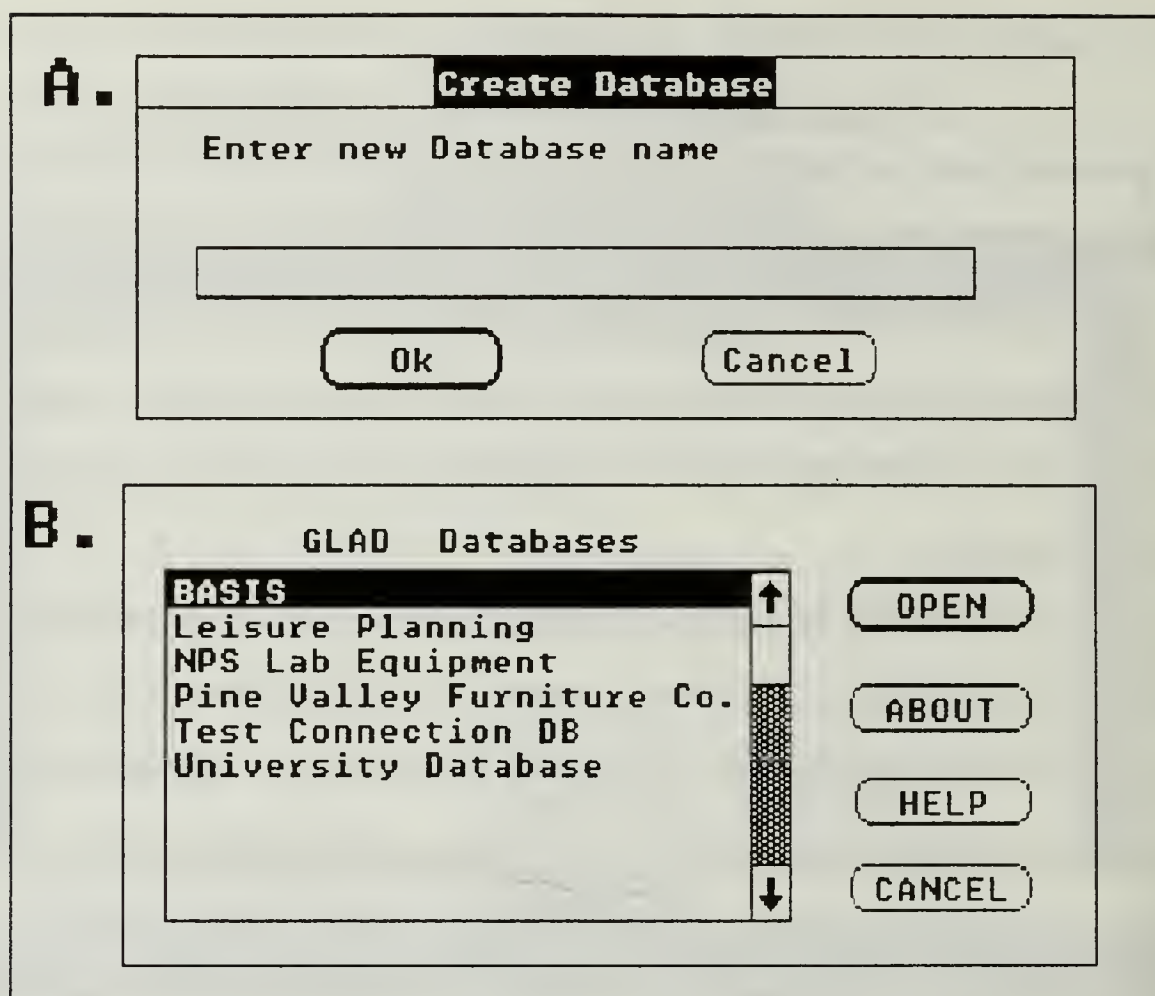
Figure 10.   Dialog Boxes for Create and Modify Functions

B. DATA DEFINITION FUNCTIONS

The initial design of the DDL facility included a child window of the main DDL window to handle the definition of entities and their attributes. The window had a separate menu and a separate control system. After initial tests with this facility, it was determined that the extra level in the hierarchy of windows made the system more complex and would probably be harder for a novice user to initially comprehend. Upon reevaluation, it was decided that all of the definition functions could be included in the main DDL menu. This reduces the number of levels of the hierarchy while keeping the length of the menu to a maximum of eight items. The final version is shown in Figure 11.

24

Figure 11. Data Definition Facility Window

The class which controls the DDL window is the DDWindow Class. The second line of the DDWindow Class is the **inherit** command which identifies the class from which the current class is inherited and all of its instance variable. The **inherit** command for the DDWindow class is reproduced below with the comments omitted; comments are enclosed by a slash and an asterisk in the class definition.

inherit(DBWindow, #DDWindow, #(name newObj nest newNest
tmpObj dMWin idx attrDialog attrList), 2, nil)!!

The first parameter in parenthesis is the class from which this class is inherited; in this case, the DBWindow class. DBWindow class is also shown in Appendix B. It contains all of the methods for painting of the window and for selecting and moving object rectangles on the screen. Since both the DDL and the DML windows use these functionss, their corresponding methods are combined into a separate higher level class. Both the DDWindow and the DMWindow class inherit the methods from the DBWindow class. In this way the same code is used for both facilities of the database, saving code, memory, and programming time. This is where the true value of Actor becomes evident.

1. Defining Entities

The Define menu item in the DDL menu allows the user to define entity objects. Figure 12 shows the dialog box which this function invokes. This dialog box is controlled by the ObjectDialog Class shown in Appendix B. In this dialog the user is asked to specify the name of the entity and whether it will be a generalization of other entities (nested) or an atomic entity. The design is such that, if no object is selected prior to selecting Define, the dialog box will be initialized with no name and with the Atomic nesting level selected. Once the Accept button is selected, the object is displayed in the center of the DDL window. The object can then be moved to the desired location using the mouse. If an object is selected (it has been highlighted with the mouse), then the dialog box is initialized with the parameters for the selected object. Any changes made to the selected object will be immediately visible in the DDL window.

Figure 12. Dialog Box for Defining Entities

2. Defining Attributes

Initially, the GLAD design called for attributes to be defined at the same time as the entity itself. The problem with this method was that user-defined

attribute types were defined before the entity to which it related was defined. This caused problems with type checking and resulted in some attributes having an attribute type which were undefined. The final version of the system separates the definition of the entities from the definition of their attributes. It requires an entity be defined before it can be used as a user-defined attribute type. The implication is that all entities should be defined prior to defining any attributes. This method results in fewer errors by allowing the system to conduct on the spot type checking. If a user tries to define an attibute that has not been defined, GLAD will issue an error message and allow the user to reenter a new attribute type.

Defining the attributes of an entity requires the user to select an object and then select the Attribute menu item. This action causes a message to be sent to the AttribDialog Class which controls the dialog box as shown in Figure 13. This dialog box is for the EMPLOYEE object of the university database. All of the previously defined attributes are displayed in the listbox at the bottom left of the dialog box. New attributes can be added by positioning the cursor to each of the input fields above the list box, entering the desired data, and selecting the Add pushbutton. Attributes can easily be deleted by selecting the attribute from the listbox and selecting Delete.

Shneiderman [Ref: 6:p. 91] suggested that error messages should rarely be needed in a good direct manipulation interface. To help the user avoid mistakes and keep error messages to a minimum, the Type List feature is included. Selecting Type List results in the display of the dialog box of Figure 14. This dialog box is controlled by the TypeDialog Class and lists all of the possible attribute types which can be used, including system-defined and user-defined types. Selecting an attribute type results in the removal of the dialog box and the insertion of the

attribute type in the Attribute Type field of the attribute dialog box. The user is not required to remember any of the available types and by being able to have the type automatically entered reduces the possibility of errors. This is especially true considering the fact the Actor is case sensitive, so Int is not equivalent to INT.

```
        Attributes for object EMPLOYEE
  Attribute Name:
  ┌─────────────────────────────────┐
  │                                 │      ╭──────────────╮
  └─────────────────────────────────┘      │     Add      │
  Attribute Type:                          ╰──────────────╯
  ┌─────────────────────────────────┐
  │                                 │      ╭──────────────╮
  └─────────────────────────────────┘      │   Delete     │
  Length of field:                         ╰──────────────╯
  ┌─────────────────────────────────┐
  │                                 │      ╭──────────────╮
  └─────────────────────────────────┘      │  Type List   │
  ┌──────────────────────────────┬──┐      ╰──────────────╯
  │Name       : String[16]       │ ↑│
  │Age        : Int[8]           │  │      ╭──────────────╮
  │Pay        : Int[8]           │  │      │    Quit      │
  │Address    : String[35]       │▓▓│      ╰──────────────╯
  │WorksFor   : DEPT[20]         │▓▓│
  │                              │▓▓│
  │                              │▓▓│
  │                              │ ↓│
  └──────────────────────────────┴──┘
```

Figure 13. Dialog Box for Defining Attributes

28

```
Int          :  S
Money        :  S
String       :  S
DEPT         :  U
EMPLOYEE     :  U
EQUIPMENT    :  U
```

[Accept]   [Cancel]
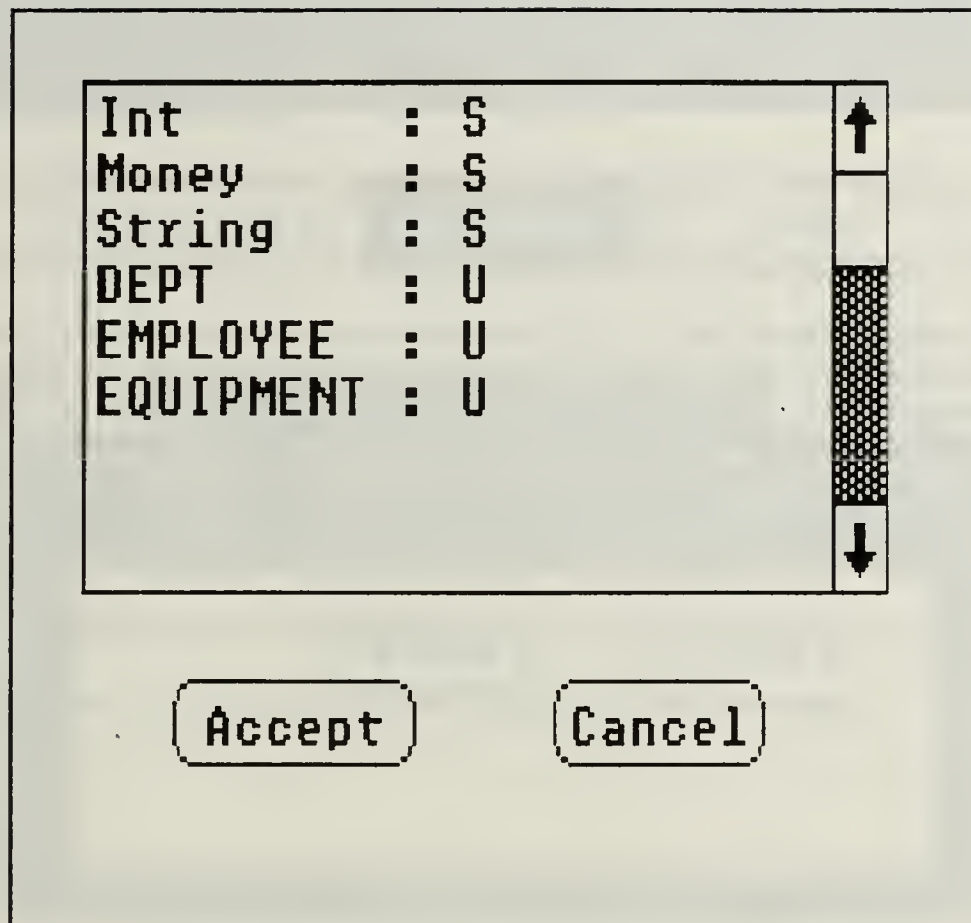
Figure 14. Type List Dialog Box

3. Expand Function

The Expand function allows the definition of specialization entities for a nested object. The expand window is the same as the main DDL window with the exception that Save and Mode are not included in the menu. This is shown in Figure 15. Since all of the remaining menu items cause the same actions as in the DDL main window menu, the Expand window is nothing more than an instance of the DDWindow Class with a shortened menu. No new code is needed for this window as all the methods needed to run the window are in the DDWindow Class. Again the power and flexibility of Actor is fully exploited.

29
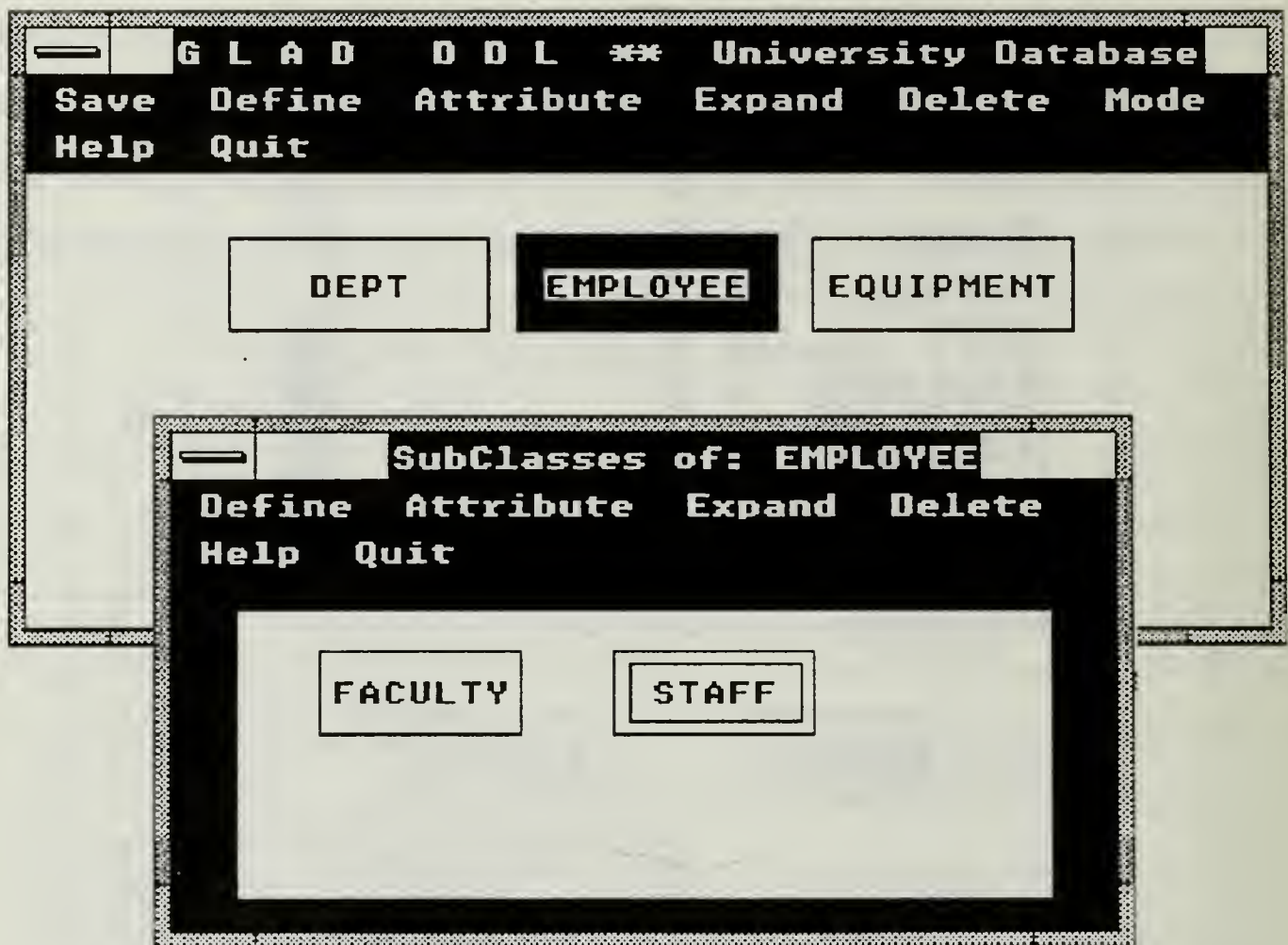
Figure 15. DDL Expansion Window

4. Other Functions

The remaining functions are fairly self-explanatory and their controlling methods are all contained in the DDWindow Class. The Save function simply saves the schema to a disk file for later use. The Delete function is used to delete entities which are no longer needed. Deleting a nested entity results in all specialized objects of that entity also being deleted.

The Mode function was primarily included as a debugging tool. The function causes the database schema currently displayed in the DML window to be displayed in the DML window, thus allowing testing of a newly created or modified database schema. This feature is especially important when modifying an existing

30

database schema. The user can evaluate the new design prior to saving it and overwriting the existing schema.

5. Error Checking

Finally, GLAD offers a full range of error checking functions. Included in these functions are checks for duplicate database and object names, type checking for attribute types, checks to ensure that changes to the database schema are saved prior to exiting, and checks to ensure all required inputs are completed prior to accepting a new entity or attributes. These errors result in a dialog box with a warning and a description of the error. These features aid in the learning process and help make the user feel comfortable to know that errors will not go undetected, nor will they result in any kind of catastrophic loss to the system.

V. CONCLUSIONS

A. DISCUSSION OF THE RESEARCH

This thesis has shown that it is possible to create a user-friendly database management system which combines the best of graphics-oriented interfaces with a data model which offers an enormous amount of flexibility over that available in popular relational models. In addition, the research has shown that this interface offers ease of use to novice users while not penalizing experienced users in the process. The virtue of this conclusion has been carried over into the realm of the data definition language. Although this function has been traditionally handled by database administrators, the GLAD system is easy enough to use that even novice users have the ability to define or modify databases. The use of Windows and Actor are instrumental in providing the facilities that make this possible by reducing the development time, as well as, the learning time of end users.

B. FUTURE PROGRAM IMPROVEMENTS

GLAD has many areas in which research still needs to be conducted. Among them are extended capabilities of current window classes, addition of full querying facility, addition of an on-line, context sensitive help screen system, system defined set type for attributes, and linking of the system to commercially available database management systems. Additionally, a means of providing secure access control to individual parts of the database must be researched before GLAD can be considered viable for Department of Defense applications. Long range goals include development of data-oriented visual programming and a unified GLAD interface to

multiple, heterogeneous databases. This must include a method of providing mutual exclusion to database elements in multitasking, networking environment.

C. BENEFITS OF RESEARCH

This research demonstrates the ease of use and the power and flexibility which can be obtained in a database management system. As the microcomputer continues to gain in popularity and number, ease of use for the end user will become increasingly important. An additional benefit can be derived from the lessons learned about object-oriented programming in Actor. The power of reusable class definitions and inheritance between classes can be exploited for many applications besides databases. Object-oriented programming is now being investigated for use in all areas of computer science. Whether the research is viewed from a programmers standpoint or an end users standpoint, the use of these systems will inevitably save time and, therefore, money.

APPENDIX A. SAMPLE USER SESSION

This sample user session is based on the database model developed for the Bases and Stations Information System (BASIS) by the Naval Data Automation Command (NAVDAC) for the Chief of Naval Operations. The top-level entity objects correspond to the database records specified by this system. Nested objects have been added where appropriate so that objects do not contain mutually exclusive attributes. For example, PERSON contains information which applies solely to officers or solely to enlisted personnel. Since one record cannot contain both types of information, these attributes are mutually exclusive and are, therefore, put into expanded objects OFFICER and ENLISTED. This is also done for FITNESS, MANPWR, and CORRES entities. FITNESS is expanded to include objects MALE and FEMALE, MANPWR is expanded to OFFICER and ENLISTED, and CORRES is broken down into LETTERS and MESSAGES.

The following is a step by step procedure for setting up the database. The term 'Select' is used to mean the cursor is placed on the selected item using the mouse and the left mouse button clicked once. This action either highlights the item or causes the menu or button function to be executed.

I.   Creation of the Database (Figure A-1)
    A.   Select Create from the GLAD top-level menu
    B.   Type 'BASIS' into the Create Database dialog box
    C.   Select 'Ok'
    Results: GLAD DDL ** BASIS window is displayed
II.   Creation of top-level entity objects (Figure A-2)
    A.   Select Define from the DDL menu
    B.   Type 'PERSON' into the OBJECT DEFINITION dialog box

C. Select Nested in the Nesting Level control box of the dialog box

D. Select Accept

E. Set location of Object by pointing at object, pressing and holding the left mouse button, and moving object to desired location. Object will move as the mouse is moved.

F. Repeat steps II. A-E for the following objects:

1. FITNESS (Physical Readiness Test data), Nested

2. CAREER (Career Counseling data), Atomic

3. TAD (Temporary Additional Duty), Atomic

4. LOCATOR (Postal Locator), Atomic

5. UA (Unauthorized Absences), Atomic

6. UIC (Unit Identification Code), Atomic

7. MANPWR (Manpower Authorization), Nested

8. DEPT (Department Table data), Atomic

9. DIV (Division Table data), Atomic

10. MUSTER (Muster Reporting), Atomic

11. ADMIN (Admin Control Record), Atomic

12. CORRES (Correspondence and Messages), Nested

13. FORMS (Forms Control), Atomic

14. RECURR (Recurring Reports), Atomic

15. INST (Instructions and Notices), Atomic

16. CHANGE (Instruction Change Transmittal), Atomic

Results: Top-Level window of Figure A-3

III. Create Expanded Objects for PERSON Object (Figure A-4)

A. Select PERSON object

B. Select Expand from DDL menu.

C. Select Define in 'Subclasses of: PERSON' window

D. Type 'OFFICER' into OBJECT DEFINITION dialog box

E. Select Atomic in the Nesting Level control box of the dialog box

F. Select Accept

G. Repeat steps C-E for ENLISTED object

Results: Window as shown in Figure A-5

IV. Create Expanded Objects for remaining nested objects

    A.   Repeat steps III. A-F above for FITNESS, MANPWR, and CORRES

V. Define Attributes for PERSON Object (Figure A-6)

    A.   Select PERSON object

    B.   Select Attribute from DDL window

    C.   Type 'SSN' into Attribute Name field

    D.   Select Attribute Type field

    E.   Type 'String' or Select Type List and select the desired attribute type

    F.   Select Length of Field

    G.   Type '9'

    H.   Select Add

    I.   Repeat Steps V. C-H for each of the attributes of PERSON (for this demonstration, only selected attributes were actually entered)

Results: Attributes as shown in Figure A-7. Select Quit to exit the attribute definition mode

VI. Define Attributes for Expanded Objects (Figure A-8)

    A.   Select Expand from DDL window

    B.   Select OFFICER object (See Figure A-5)

    C.   Select Attribute from 'Subclasses of: PERSON' window

    D.   Repeat V. C-I for all attributes of OFFICER

Results: Attributes entered as shown in Figure A-8. Select Quit to exit Subclasses window

VII. Define Attributes for Remaining Objects

    A.   Repeat steps V. A-I for each remaining top-level object

    B.   Repeat steps VI. A-D for each expanded object

VIII Save and Exit the DDL routine

    A.   Select Save from DDL top-level menu

    B.   Select Quit

IX. Actual data can now be entered using the change function of the Data Manipulation Language.
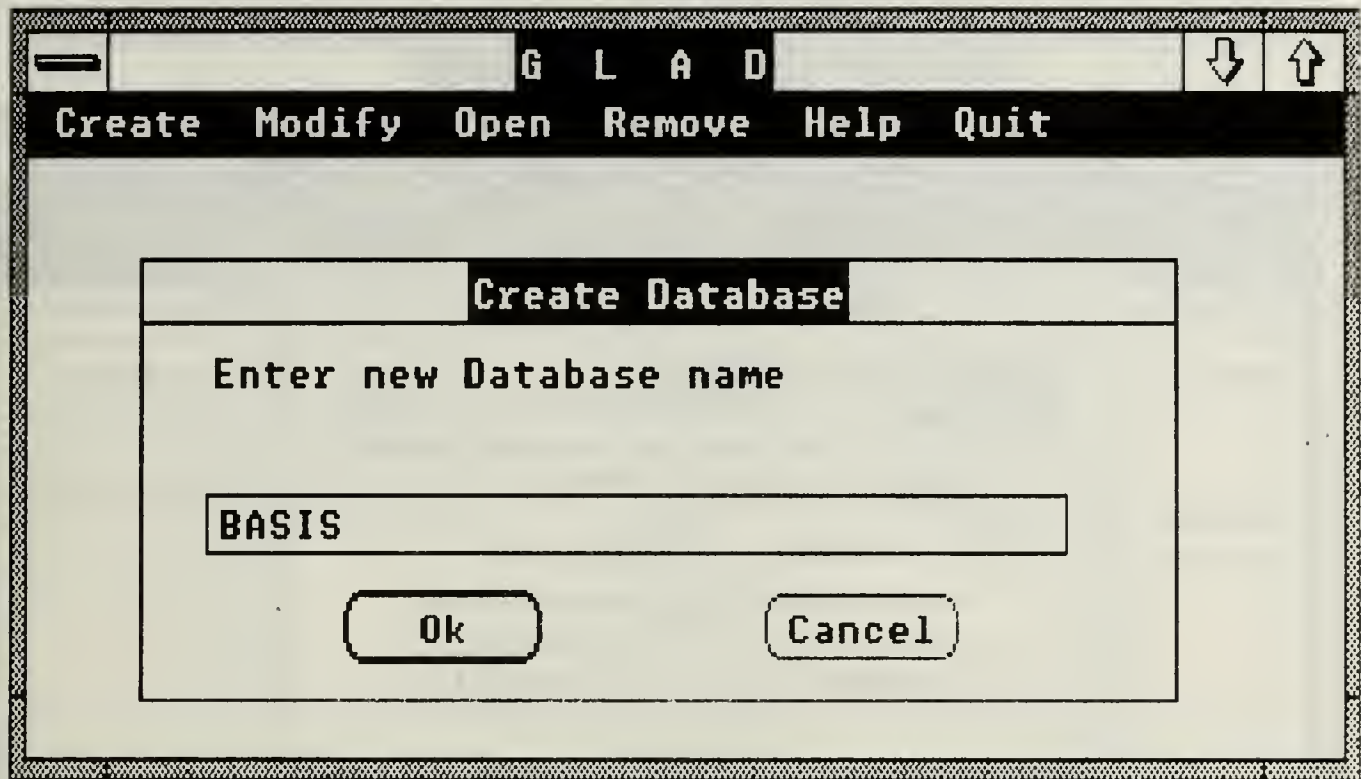
FIGURE A-1 BASIS DDL Window

Figure A-2 Define **PERSON** Entity

38
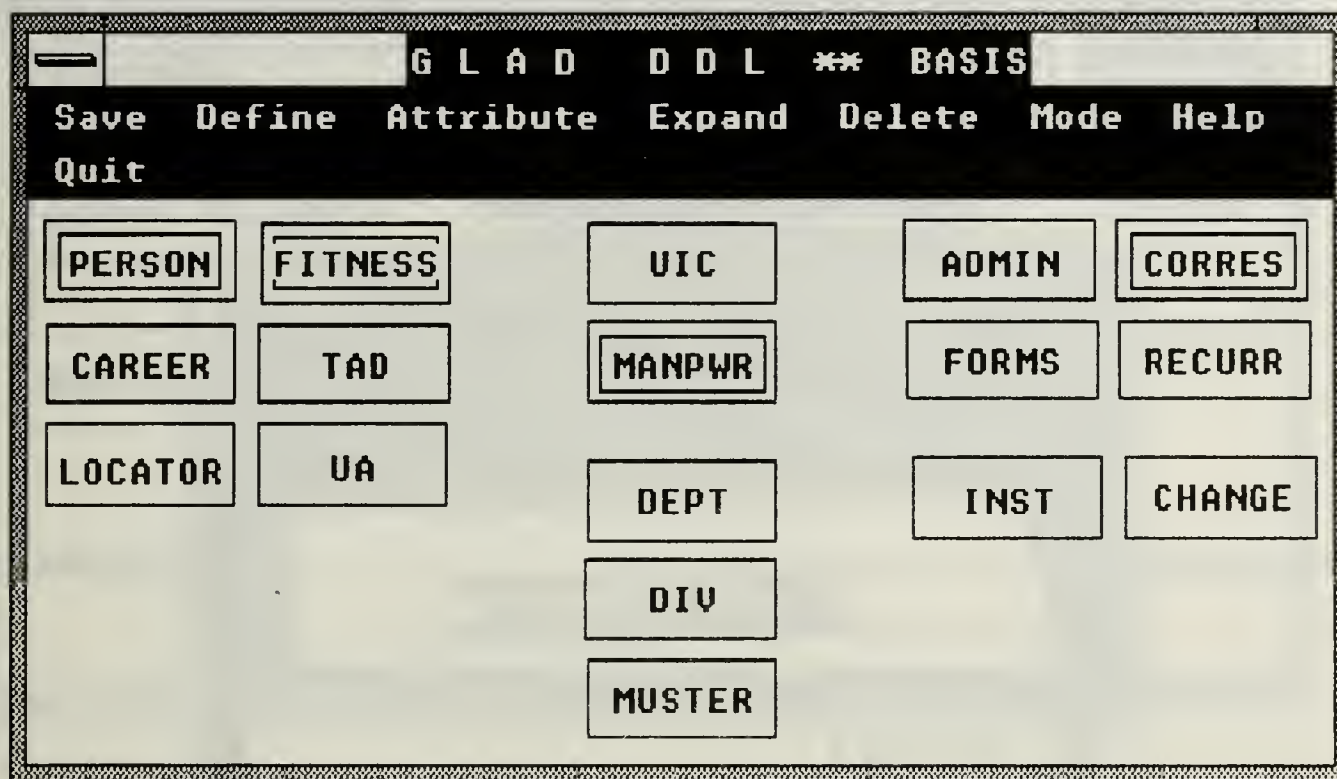
Figure A-3 BASIS Top-Level Window

```
┌──────────────────────────────────────────────────────────┐
│ ▬         G L A D   D D L  **  BASIS                      │
│ Save  Define  Attribute  Expand  Delete  Mode  Help       │
│ Quit                                                      │
│                                                           │
│ ┌───────┐ ┌───────┐    ┌───────┐    ┌───────┐ ┌───────┐  │
│ │PERSON │ │FITNESS│    │  UIC  │    │ ADMIN │ │CORRES │  │
│ └───────┘ └───────┘    └───────┘    └───────┘ └───────┘  │
│                                                           │
│ ┌───────┐ ┌───────┐  ┌───────┐      ┌───────┐ ┌───────┐  │
│ │CAREER │ │  TAD  │  │MANPWR │      │ FORMS │ │RECURR │  │
│ └───────┘ └───────┘  └───────┘      └───────┘ └───────┘  │
│         ┌─────────────────────────────────────────┐      │
│ ┌─────┐ │ ▬        SubClasses of: PERSON          │─┐    │
│ │LOCATO│ │ Define  Attribute  Expand  Delete      │GE│    │
│ └─────┘ │ Help  Quit                              │─┘    │
│         │                                         │      │
│         │      ┌──────────────────┐               │      │
│         │      │OBJECT DEFINITION │               │      │
│         │      └──────────────────┘               │      │
│         │      Enter Object Name:                 │      │
│         │      ┌──────────────────────────────┐   │      │
│         │      │OFFICER                       │   │      │
│         │      └──────────────────────────────┘   │      │
│         │      ┌─Nesting Level────────────┐       │      │
│         │      │ ◉ Atomic    ○ Nested     │       │      │
│         │      └──────────────────────────┘       │      │
│         │      ( Accept )      ( Cancel )          │      │
│         └─────────────────────────────────────────┘      │
└──────────────────────────────────────────────────────────┘
```

Figure A-4   Create Expanded Objects for PERSON Object

```
┌─────────────────────────────────────────────────────────────┐
│ ▭           G L A D    D D L  ** BASIS                        │
│ Save  Define  Attribute  Expand  Delete  Mode  Help          │
│ Quit                                                         │
├─────────────────────────────────────────────────────────────┤
│ ┌────────┐ ┌────────┐      ┌────────┐    ┌────────┐ ┌────────┐│
│ │ PERSON │ │FITNESS │      │  UIC   │    │ ADMIN  │ │ CORRES ││
│ └────────┘ └────────┘      └────────┘    └────────┘ └────────┘│
│ ┌────────┐ ┌────────┐    ┌────────┐      ┌────────┐ ┌────────┐│
│ │ CAREER │ │  TAD   │    │ MANPWR │      │ FORMS  │ │ RECURR ││
│ └────────┘ └──┌──────────────────────────────────┐─┘ └────────┘│
│ ┌────────┐    │ ▭        SubClasses of: PERSON    │           │
│ │ LOCATO │    │ Define  Attribute  Expand  Delete │  CHANGE   │
│ └────────┘    │ Help  Quit                        │           │
│               │  ┌──────────┐   ┌──────────┐      │           │
│               │  │ OFFICER  │   │ ENLISTED │      │           │
│               │  └──────────┘   └──────────┘      │           │
│               │                                   │           │
│               │                                   │           │
│               └───────────────────────────────────┘           │
└─────────────────────────────────────────────────────────────┘
```
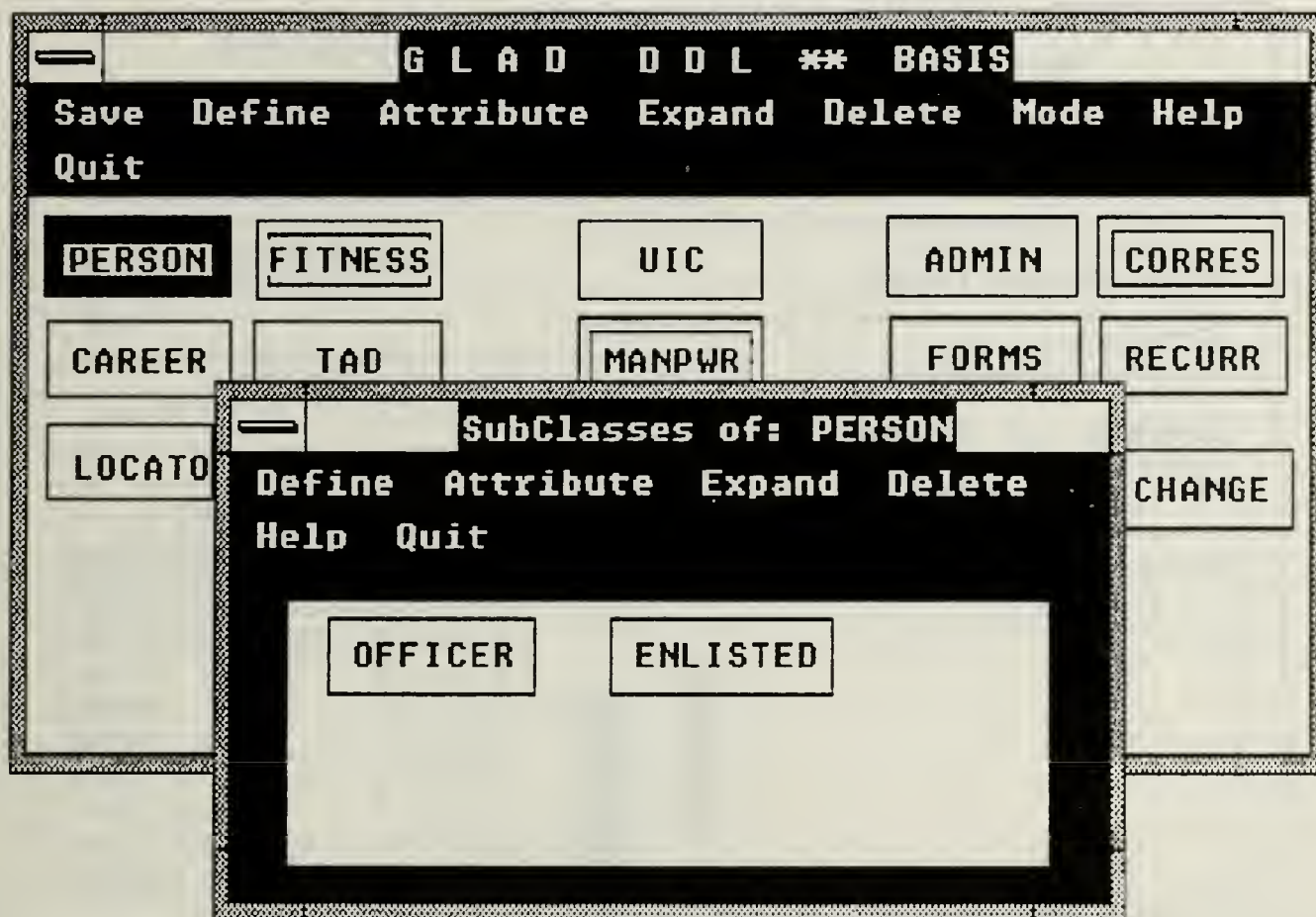
Figure A-5   Expanded Objects for PERSON Object

41

```
+-----------------------------------------------------------------+
|              Attributes for object PERSON                       |
|  Attribute Name:                                                |
|  +---------------------------------------+                      |
|  | SSN                                   |                      |
|  +---------------------------------------+    .------------.    |
|                                               (    Add     )    |
|  Attribute Type:                              '------------'    |
|  +---------------------------------------+                      |
|  | String                                |    .------------.    |
|  +---------------------------------------+    (   Delete   )    |
|                                               '------------'    |
|  Length of field:                                               |
|  +---------------------------------------+    .------------.    |
|  | 9                                     |    ( Type List  )    |
|  +---------------------------------------+    '------------'    |
|  +-----------------------------------+---+                      |
|  |                                   | ↑ |    .------------.    |
|  |                                   +---+    (    Quit     )   |
|  |                                   |   |    '------------'    |
|  |                                   |###|                      |
|  |                                   |###|                      |
|  |                                   |###|                      |
|  |                                   +---+                      |
|  |                                   | ↓ |                      |
|  +-----------------------------------+---+                      |
+-----------------------------------------------------------------+
```

Figure A-6   Define Attributes for PERSON Object

42

```
          Attributes for object PERSON
   Attribute Name:
  ┌─────────────────────────────────┐
  │ ACCOUNT_CAT_CODE                │
  └─────────────────────────────────┘        ╭──────────────╮
                                              ( Add          )
   Attribute Type:                            ╰──────────────╯
  ┌─────────────────────────────────┐
  │ Int                             │
  └─────────────────────────────────┘        ╭──────────────╮
                                              ( Delete        )
   Length of field:                           ╰──────────────╯
  ┌─────────────────────────────────┐
  │ 3                               │
  └─────────────────────────────────┘        ╭──────────────╮
                                              ( Type List     )
  ┌──────────────────────────┬──┐             ╰──────────────╯
  │ ACCOUNT_CA: Int[3]       │ ↑│
  │ ACT_DUTY_S: Date[6]      │  │             ╭──────────────╮
  │ ACTUAL_DET: Date[6]      │  │             ( Quit          )
  │ DATE_RANK_: Date[6]      │  │             ╰──────────────╯
  │ NAME     : String[27]    │  │
  │ DEPT_CODE_: DEPT[6]      │  │
  │ DIV_CODE_A: DIV[6]       │  │
  │ PAY_GRADE : String[4]    │  │
  │ ACCESS_LEV: String[1]    │  │
  │ SERVICE_BR: String[1]    │ ↓│
  └──────────────────────────┴──┘
```

Figure A-7   Attributes for PERSON Object

43

```
      Attributes for object OFFICER
   Attribute Name:
   ┌──────────────────────────────┐
   │                              │
   └──────────────────────────────┘
                                        ╭───────────────╮
                                        │      Add      │
   Attribute Type:                      ╰───────────────╯
   ┌──────────────────────────────┐
   │                              │
   └──────────────────────────────┘     ╭───────────────╮
                                        │    Delete     │
   Length of field:                     ╰───────────────╯
   ┌──────────────────────────────┐
   │                              │     ╭───────────────╮
   └──────────────────────────────┘     │   Type List   │
                                        ╰───────────────╯
   ┌────────────────────────────┬─┐
   │DESIGNATOR: Int[4]          │↑│
   │LINEAL_NUM: String[9]       ├─┤     ╭───────────────╮
   │RANK      : String[5]       │ │     │     Quit      │
   │                            │▓│     ╰───────────────╯
   │                            │▓│
   │                            │▓│
   │                            │▓│
   │                            ├─┤
   │                            │↓│
   └────────────────────────────┴─┘
```

Figure A-8 Attributes for Subclass OFFICER

APPENDIX B. SOURCE CODE LISTING

The following is a list of those classes which were created or modified specifically for the implementation of the Data Definition Language. The classes which were modified from previous versions of GLAD are indicated after the class title.

1. AttribDialog Class

```
/* Dialog for entering entity object attributes  */!!

inherit(Dialog, #AttribDialog, #(attrList  /* List of
attributes in the same format as selObj.attributes */
selAttr  /* Attribute selected from listbox */
objName /*Name of the object for which these attributes
  apply*/
typeList /* List of possible types of attributes including
  system and user defined attributes */
typeDialog /* Type dialog box */
tempArray /* Temporary attribute array*/
obj /*Array of two elements for setting up tempArray */
), 2, nil)!!

now(AttribDialogClass)!!

/* Creates a new instance of the class AttribDialog  */
Def  new(self,selObj | theDlg)
{
  theDlg := new(self:Behavior);
  theDlg.objName := selObj.name;
  theDlg.selAttr := new(String, 15);
  if selObj.attributes
    theDlg.attrList := selObj.attributes
  else
    theDlg.attrList := new(OrderedCollection, 15)
  endif;
  ^theDlg
```

45

```
}    !!

now(AttribDialog)!!

/* Searches type list for valid type names */
Def  isTypeDef(self, aColl, newName)
{
  do(aColl,
    {using(elem) if elem[0] = newName
    ^elem[1]
    endif;});
  ^nil
}    !!

/* Inserts a new attribute string into the list box of the
   AttribDialog */
Def insertString(self,aStr | ans, insertLoc)
{
  if selAttr
   insertLoc := selAttr
  else
   insertLoc := -1
  endif;

  ans := Call SendDlgItemMessage(hWnd, ATTR_LIST,
              LB_INSERTSTRING,insertLoc,
              lP(aStr));
  freeHandle(aStr);
  ^ans
}     !!

Def addString(self,aStr | ans)
{
  ans := Call SendDlgItemMessage(hWnd, ATTR_LIST,
              LB_ADDSTRING,0,lP(aStr));
  freeHandle(aStr);
  ^ans
}   !!

/* This method adds a new attribute to the selected objects
   attribute list */
Def addAttr(self)
{  tempArray := new(Array, 4);
```

46

```
        tempArray[0] := getItemText(self, ATTR_NAME);
        tempArray[1] := getItemText(self, OBJ_ATTRIB);
        tempArray[2] := getItemText(self, ATTR_LENGTH);
        if (tempArray[0] = "" or tempArray[1] = "" or
           tempArray[2] = "")
          errorBox("ERROR!", "Fill in all attribute fields.")
        else
          if isTypeDef(self, attrList, tempArray[0])
            errorBox("WARNING!", "Attribute already exists.")
          else
            if tempArray[3] := isTypeDef(self, typeList,
                        tempArray[1])
            selAttr := insertString(self,
subString(tempArray[0] + "        ", 0, 10)
                + ": " + tempArray[1] + "[" + tempArray[2] +
                "]");
            insert(attrList, tempArray, selAttr);
            setCurSel(self, selAttr);
            initEditBox(self)
          else
            errorBox("WARNING!", "Invalid Attribute Type.")
          endif
        endif
      endif
}                    !!

/* Clears the text edit boxes of the dialog. */
Def  initEditBox(self)
{
 setItemText(self, ATTR_NAME,"");
 setItemText(self, OBJ_ATTRIB, "");
 setItemText(self, ATTR_LENGTH, "");
 selAttr := nil
}   !!

/* This method deletes attributes from an object's
   attribute list */
Def  deleteAttr(self)
{
 if selAttr
   remove(attrList, selAttr);
   Call SendDlgItemMessage (hWnd,ATTR_LIST,
              LB_DELETESTRING,
```

47

```
                    selAttr,0);
        initEditBox(self);
        setCurSel(self, -1);
        selAttr := nil
    else
        errorBox("WARNING!", "Select Attribute to delete.")
    endif
}      !!

/* Selects attribute in list box and initializes the
   variables and edit boxes */
Def  selItem(self)
{
  selAttr := Call SendDlgItemMessage (hWnd,ATTR_LIST,
        LB_GETCURSEL,0,0);
  if (selAttr >= 0 and selAttr < size(attrList))
    setItemText(self, ATTR_NAME,attrList[selAttr][0]);
    setItemText(self, OBJ_ATTRIB, attrList[selAttr][1]);
    setItemText(self, ATTR_LENGTH, attrList[selAttr][2])
  else
    initEditBox(self)
  endif;

}      !!

/*set the current selection to idx,  Used for highlighting
   the selected item in the list box. */
Def setCurSel(self, idx)
{
  ^Call
SendDlgItemMessage(hWnd,ATTR_LIST,LB_SETCURSEL,idx,0)
}      !!

Def  command(self, wP, lP)
{
  select
    case wP == IDCANCEL
      is setAttrList(parent, attrList);
        end(self, 0)
    endCase

    case wP == ATTR_DELETE
      is deleteAttr(self)
```

48

```
      endCase

      case wP == ATTR_TYPE
        is typeDialog := new(TypeDialog, typeList);
          runModal(typeDialog, ATTRLIST, self)
      endCase

      case wP == IDOK
        is addAttr(self);
          setItemFocus(self, ATTR_NAME);
      endCase

      case wP == ATTR_LIST
        is selItem(self)
      endCase
    endSelect

}                 !!

/* Initialize the listbox;  the method is the Actor
   equivalent of WM_INITDIALOG */
Def  initDialog(self, wP, lP)
{
  selAttr := nil;
  setItemText(self, OBJ_NAME, objName);
  do(attrList,
    {using(elem) insertString(self, subString(elem[0]
              + "         ", 0,10) + ": "
              + elem[1] + "[" +
              elem[2] + "]")});
  typeList := new(OrderedCollection, 15);
  add(typeList, tuple("Date", "S"));
  add(typeList, tuple("Int", "S"));
  add(typeList, tuple("Money", "S"));
  add(typeList, tuple("String", "S"));
  typeList := getDispObj(parent, typeList);
}                 !!
```

2. ColorTable Class (Modified)

/* collection of colors available for shading Glad objects

```
      */!!

inherit(OrderedCollection, #ColorTable, nil, 2, 1)!!

now(ColorTableClass)!!

now(ColorTable)!!

/*initialize the table with
  colors available in the system */
Def set(self | elem, colors)
{
 /*first get the available colors*/
 /*getColorTable() is in ACT dir*/
 colors := getColorTable();
 do(colors, {using(color)
          if color <> 0 and color <> WHITE_COLOR
           /*dont add BLACK or WHITE*/
           elem := new(Array,2);
           elem[USED] := nil;
           elem[COLOR] := color;
           add(self,elem)
          endif})
}


    !!

/*returns the next available color for shading*/
Def nextBrushColor(self)
{
 do(self, {using(elem)
          if not(elem[USED])
            elem[USED] := true;
            ^elem[COLOR]
            endif});
 errorBox("E R R O R","No more available color");
 ^WHITE_COLOR
}    !!

/*makes the unselected object's color available again*/
Def avail(self, color)
```

```
{
  do(self, {using(elem)
          if elem[COLOR] = color
           elem[USED] := nil;
          ^0
          endif})
}    !!
```

3. DBDialog Class (Modified)

```
/* This dialog list the databases currently available
   under GLAD.  The dialog uses either OPNDBLIST or
   RMVDBLIST dialog template depending on the state.
   Buttons HELP, OPEN, REMVOVE and CANCEL are self
   explanatory. ABOUT describes the selected database. */!!

inherit(Dialog, #DBDialog, #(dbNames /* collection of
  databases names used for listing purpose */
rmvDbNames/*collection of databases selected to be removed*/
state /*tells whether the dialog is in Open or Remove mode*/
selDb /* selected db to be opened  */
), 2, nil)!!

now(DBDialogClass)!!

Def new(self | theDlg, gladDbs, line)
{
  theDlg := new(self:Behavior);
  theDlg.dbNames    := new(SortedCollection,15);
  theDlg.rmvDbNames  := new(SortedCollection,15);
  theDlg.selDb       := new(String,30);

  gladDbs := new(TextFile);
  setName(gladDbs, "glad.dbs");
  open(gladDbs,0); /* 0 means read-only */

  loop while (line := readLine(gladDbs)) begin
    add(theDlg.dbNames, line)
  endLoop;

  close(gladDbs);
```

51

```
    ^theDlg
}    !!

now(DBDialog)!!

/* Adds a newly created DB name to the dbList */
Def  addDb(self, db)
{
  if not(find(dbNames, db))
    add(dbNames, db)
  endif;
}   !!

/*gets the filename from the name listed in the listbox*/
Def fileNameOfSelDb(self | tmpStr)
{
  tmpStr := new(String,30);
  tmpStr := "";
  do(selDb, {using(elem)
        if elem <> ' '
          tmpStr := tmpStr + asString(elem)
        endif });
  ^subString(tmpStr,0,7) + ".sch"
}         !!

/* Return  the selected database name to the calling method.
   */
Def getSelDb(self)
{
  ^selDb
} !!

/* if confirmed, then remove the selected db from
   dbNames and temporarily save it in rmvDbNames */
Def rmvDbFrom(self ,text)
{
  if questionBox("Are you sure?",
          "Really remove"+CR_LF+text) == IDYES
    add(rmvDbNames, text);
    remove(dbNames, text)
  endif
}     !!
```

```
/*update the glad.dbs file if any db is removed*/
Def updateDbsFile(self | gladDbs)
{
  if rmvDbNames   /*some db is removed */
    do(rmvDbNames,
       {using(elem | result)
         result :=
           questionBox("W A R N I N G",
                    "Really remove " + elem + "?");
         if result == IDNO  /* then restore it */
           add(dbNames,elem)
         endif
       } )
  endif;

  gladDbs := new(TextFile);
  setName(gladDbs,"glad.dbs");
  create(gladDbs);
  do(dbNames, {using(elem) write(gladDbs,elem+CR_LF)});
  close(gladDbs)
}   !!

/*set the current selection to idx */
Def getSelIdx(self)
{
  ^Call SendDlgItemMessage(hWnd,DB_LB,LB_GETCURSEL,0,0)
}  !!

Def command(self,wP,lP)
{
  select
    case wP == IDCANCEL
    is end(self,0)
    endCase

    case wP == ABOUT_DB
    is
       if (selDb := getLBText(self,DB_LB) ) /*not nil*/
         errorBox("ABOUT","brief description "+selDb)
       else
         errorBox("E R R O R!?!",
               "Database was not selected properly")
       endif
```

```
      endCase

      case wP == HELP_LB
      is errorBox("Help","discuss other buttons");
      endCase

      /* selection was made and double-clicked
         or DEFBUTTON (either Open or Remove)
         was pressed  */
      case (wP == DB_LB and high(lP) = LBN_DBLCLK)
           or (wP == DEFBUTTON)
      is
        if (selDb := getLBText(self,DB_LB)) /*not nil*/
          if state == REMOVE_DB
            rmvDbFrom(self,selDb)
          endif;
          end(self,state)
        else
          errorBox("E R R O R!?!",
                "Database was not selected properly");
        ^1
        endif
      endCase
    endSelect;
    ^1
}       !!

/*set the current selection to idx */
Def setCurSel(self, idx)
{
 ^Call SendDlgItemMessage(hWnd,DB_LB,LB_SETCURSEL,idx,0)
} !!

/* Adds a string to the DB listbox */
Def addString(self,aStr | ans)
{
 ans := Call SendDlgItemMessage(hWnd, DB_LB,
                LB_ADDSTRING,0,lP(aStr));
 freeHandle(aStr);
 ^ans
} !!

/*initialize the listbox, the method is the Actor
```

```
    equivalent of WM_INITDIALOG */
Def initDialog(self,wP,lP)
{
  do (dbNames,
      {using(elem) addString(self,elem)});
  setCurSel(self,0)
}    !!
```

4. DBSchema Class (Modified)

```
/* file containing a database schema */!!

inherit(TextFile, #DBSchemaFile, #(refNumber /* Number to
  reference object within the ordered collection */
), 2, nil)!!

now(DBSchemaFileClass)!!

now(DBSchemaFile)!!

/* saves the attributes to the specified file. */
Def saveAttr(self, attr)
{
  do (attr,
      {using(elem)
        write(self, elem[0] + "&" + elem[1] + "&" +
              elem[2] + "&" + elem[3] + "&" +
              CR_LF)});
  write(self, "&&" + CR_LF);
}    !!

/* Saves object and its attributes to a file. */
Def saveObj(self, object)
{
  write(self, object.name + CR_LF);
  write(self, asString(x(object.pt)) + "@" +
          asString(y(object.pt)) + CR_LF);
  saveAttr(self, object.attributes);
  if object.memberFile <> ""
    write(self, object.memberFile + CR_LF)
  else
```

```
      write(self, subString(object.name + "        ", 0, 7)
          + ".dat" + CR_LF)
    endif;
    if object.nesting
      write(self, "G" + CR_LF);
      saveDbSchema(self, object.nesting);
      write(self, "@@" + CR_LF)
    else
      write(self, "N" + CR_LF)
    endif;

}      !!

/* Saves schema to file. */
Def  saveDbSchema(self, aColl | idx)
{
  idx := 0;
  loop
  while (idx < aColl.lastElement)
    saveObj(self, aColl[idx]);
    idx := idx + 1
  endLoop;
}  !!

/*get the attributes for the object*/
Def getAttr(self | aColl, anAttr, aStr)
{
  aColl := new(OrderedCollection,10);
  loop
  while ( (aStr := readLine(self)) <> "&&" )
    anAttr := new(Array,4);
    do (over(NAME,USER_DEF+1),
      {using(idx)
        anAttr[idx] :=
            subString(aStr,0,indexOf(aStr,'&',0));
        aStr := delete(aStr,0,size(anAttr[idx])+1)});
    add(aColl,anAttr)
  endLoop;
  ^aColl
}    !!

/*gets the schema info, stored them in an ordered
  collection and return it*/
```

```
Def getSchema(self | schemaColl, aGladObj)
{
 schemaColl := new(OrderedCollection,10);
 loop
 while (aGladObj := nextObj(self))
   add(schemaColl,aGladObj);
 endLoop;
 ^schemaColl
}    !!

/*gets the next object from the schema file*/
Def nextObj(self | tmpObj)
{
 tmpObj := new(GladObj);
 if ( (tmpObj.name := readLine(self)) <> "@@")
   /*there's more*/
   tmpObj.pt := asPoint(readLine(self));
   tmpObj.attributes:= getAttr(self);
   tmpObj.memberFile:= readLine(self);
   if (at(readLine(self),0)=='G')
    tmpObj.nesting := getSchema(self)
   endif;
   tmpObj.refCnt := 0;
   tmpObj.color := WHITE_COLOR; /*unselected color*/
   /*more fields later*/
   ^tmpObj
 else
   ^nil
 endif
}   !!
```

5. DBWindow Class

/* GLAD Window for data manipulation interaction */!!

```
inherit(Window, #DBWindow,
#(dbSchema /*meta data of opened db*/
prevObj /*previously selected object if any */
selObj  /*currently selected object if any*/
colorTable /*available colors for shading*/
hDC    /*display context*/
```

```
offset  /*difference expressed as point between the origin
  of box and mouse position*/
rbuttonDn /*state of right button*/
objMoved /*true if object is dragged*/
menuID  /*ID of menu directory*/
), 2, nil)!!

now(DBWindowClass)!!

now(DBWindow)!!

/* Initializes dbSchema for a new database */
Def  initSchema(self)
{
  dbSchema := new(OrderedCollection, 10)
} !!

/*initialize the color table.  This method
  is called from the new method*/
Def init(self)
{ initMenuID(self);
  colorTable := new(ColorTable,10);
  set(colorTable)
}
        !!

/*move the object*/
Def drag(self,wp,point | aLPt)
{
  if selObj
    objMoved := true;
    setup(self,hDC);
    eraseRect(selObj,hDC);
    aLPt := logicalPt(self,point);
    setNewOriginPt(selObj,aLPt,offset);
    display(selObj,hDC)
  endif
}        !!

/* Handles Windows event right mouse button up */
Def WM_RBUTTONUP(self,wp,lp | tmpObj)
{
  if not(rbuttonDn)
```

```
   ^0
endif;
rbuttonDn := nil;
Call ReleaseCapture();
tmpObj := objSelected(self,logicalPt(self,asPoint(lp)));
if tmpObj /*an object is clicked with rbutton*/
 if tmpObj <> selObj
  if tmpObj.color = WHITE_COLOR
    errorBox("Wrong Button??",
        "Use LEFT button to select an object")
  else
    errorBox("E R R O R",
        "RIGHT button clicked object is not"+CR_LF+

        "the selected (bold-lined) object")
  endif
 else /* = selObj */
  if selObj.aDscrbWin
    Call DestroyWindow(handle(selObj.aDscrbWin))
  endif;
  if selObj.aLMWin
    Call DestroyWindow(handle(selObj.aLMWin))
  endif;
  if selObj.aOMWin
    Call DestroyWindow(handle(selObj.aOMWin))
  endif;
  if selObj.aNDMWin
    Call DestroyWindow(handle(selObj.aNDMWin))
  endif;

  if selObj.refCnt = 0
    /*unshade it if not referenced by other objects*/
    avail(colorTable,selObj.color);
    selObj.color := WHITE_COLOR
  endif;
  /*now unselect it*/
  selObj.thickBorder := nil;
  setup(self,hDC);
  display(selObj,hDC)
 endif
endif;
selObj := nil;
releaseContext(self,hDC)
```

```
}
      !!

/* Handles Windows event right mouse button down */
Def WM_RBUTTONDOWN(self,wp,lp)
{
  if rbuttonDn
   ^0
  endif;
  rbuttonDn := true;
  Call SetCapture(hWnd);
  hDC := getContext(self)
} !!

Def command(self,wp,lp)
{/*only interprets the menu choice now*/
  if menuID[wp]
    perform(self,menuID[wp])
  else
   ^0
  endif
}  !!




/*decribes other DML commands*/
Def help(self)
{
  errorBox("H E L P","at your service")
} !!




/*setup the display context*/
Def setup(self, hdc | aRect, wd, ht)
{
  Call SetMapMode(hdc,MM_ANISOTROPIC);
  aRect := clientRect(self);
  wd    := width(aRect);
  ht    := height(aRect);
  Call SetWindowExt(hdc,1024,512);
  Call SetViewportExt(hdc,wd,ht)
}  !!
```

```
/*left button is released*/
Def endDrag(self,wp,pointl aLPt)
{
  select
    case selObj and not(objMoved)
    /*an object is selected and was not moved*/
    is if prevObj /*unbold the bolded border*/
        if not(prevObj.aDscrbWin or prevObj.aLMWin
            or prevObj.aOMWin or prevObj.aNDMWin)
          and prevObj.refCnt=0
        /*also unshade since it has no DWin,OMWin,LMWin
          and is not referenced by other objects*/
          avail(colorTable,prevObj.color);
          prevObj.color := WHITE_COLOR
        endif;
        prevObj.thickBorder:= nil;
        display(prevObj,hDC)
      endif;
      if selObj.color = WHITE_COLOR
        /*not referenced in another's describe window,
          so assign it a color*/
        selObj.color := nextBrushColor(colorTable)
      endif;
      selObj.thickBorder := true;
      display(selObj,hDC)
    endCase

    case selObj and objMoved
    /*an object is just moved, so unselect it*/
    is
      display(selObj,hDC);
      selObj := prevObj
    endCase
  endSelect;
  releaseContext(self,hDC);
  repaint(self)
}          !!

/*left button is pressed; check if the cursor is within
  the object rectangle. If yes get ready to move or
  select it*/
Def beginDrag(self,wp,point I aLPt)
{
```

```
  aLPt := logicalPt(self,point);
  objMoved := nil;
  if selObj
    /*remember it if some object is currently selected*/
    prevObj := selObj
  endif;
  if (selObj := objSelected(self,aLPt))
    offset := getOffset(selObj,aLPt)
  endif;
  hDC := getContext(self);
  setup(self,hDC)
}           !!


/*converts the client coordinate pt to a logical pt*/
Def logicalPt(self,aClientPt | aLogiPt,aRect,wd,ht)
{
  aRect := clientRect(self);
  wd    := width(aRect);
  ht    := height(aRect);
  aLogiPt := new(Point);
  aLogiPt.x := aClientPt.x * 1024 /wd;
  aLogiPt.y := aClientPt.y * 512 /ht;
  ^aLogiPt
}  !!


/*detects whether the cursor is in the object rect*/
  /*everything is in a logical coordinate*/
Def objSelected(self,cursorPt)
{
  do (dbSchema,{using(obj)
          if containedIn(obj,cursorPt)
              ^obj /*return the selected obj*/
          endif });
  ^nil
}   !!


/*draws the diagram*/
/*this paint method is called by the show method
  via update method which sends WM_PAINT */
Def paint(self, hdc)
{
  /*set the mode*/
  setup(self,hdc);
```

```
/*display objects*/
do (dbSchema, {using(obj) display(obj,hdc)})
}  !!
/*gets the meta data of db to be opened
and initialize other instance variables*/
Def loadSchema(self, aSchemaFile | aFile)
{
  aFile := new(DBSchemaFile);
  setName(aFile,aSchemaFile);
  open(aFile,0); /*read-only*/
  dbSchema := getSchema(aFile);
  close(aFile)
}    !!
```

6. DDWindow Class

/* GLAD Window for data definition interaction */!!

inherit(DBWindow, #DDWindow, #(name /* Contains name of new
  object */
newObj /* Instance of new dialog box*/
nest /* nesting level of object */
newNest /*used to hold new nesting level if changed*/
tmpObj /* new Object */
dMWin /* Instance of DML Window */
idx /* Index reference within ordered collection */
attrDialog /*Instance of Attribute dialog box */
attrList /*List of possible attributes */
), 2, nil)!!

now(DDWindowClass)!!

now(DDWindow)!!

```
/* Ensures that changes are saved prior to closing
   the DDL window. */
Def  closeDD(self)
{
   if (changed and expanded <= 0)
     if new(ErrorBox, self, "Do you wish to save the
         changes?", "Warning - Schema not saved",
```

```
        MB_YESNO) == IDYES
     saveObj(self)
   endif
  endif;
  expanded := expanded - 1;
  close(self)
}   !!


/* Enters all displayed objects into the list of objects,
   called objList, which can be used as attribute types. */
Def  getDispObj(self , objList)
{
 do(dbSchema, {using(elem)
          add(objList, tuple(elem.name,"U"));}) ;
 ^objList
}       !!


/* Sets selected object attribute list to new list from
   changes made in the attribute dialog. */
Def  setAttrList(self, attrList)
{
  changed := true;
  selObj.attributes := attrList;
}    !!


/* Expands a nested object for definition or modification of
an objects nested entities. */
Def expandObj(self )
{
 if not(selObj)
   errorBox("ERROR!","No object is selected")
 else
  if not(selObj.nesting)
    errorBox("ERROR!","Selected object is not nested")
  else
   expanded := expanded + 1;
   selObj.aNDMWin :=
          new(NestDDWindow,self,"DdlNestMenu",
            "SubClasses of: "+ selObj.name,nil);
   start(selObj.aNDMWin,selObj,
       colorTable)
  endif
 endif
```

```
}         !!

/* Calls the attribute definition dialog box for defining
   and changing the selected objects attributes. */
Def  attrDef(self)
{
  changed := true;
  if not(selObj)
    errorBox("WARNING!", "No Object Selected")
  else
    attrDialog := new(AttribDialog, selObj);
    runModal(attrDialog, ATTRIB, self);
  endif
}              !!

/* Used for modification of an existing object */
Def  nameObj(self)
{
  if not(selObj)
    errorBox("No Object Selected",
        "Select an object prior to menu selection.");
    ^0
  endif;

  tmpObj := new(GladObj);
  if selObj.nesting
    nest := NESTED
  else
    nest := ATOMIC
  endif;

  newObj := new(ObjectDialog, selObj.name, nest);
  if runModal(newObj, DEFOBJ, self) == IDOK
    newNest := getNestLevel(newObj);
    defineNewObject(tmpObj, getText(newObj),
            newNest);
    if isNameDefined(tmpObj, dbSchema) == nil or
        nest <> newNest
      selObj.name := tmpObj.name;
      selObj.nesting := tmpObj.nesting;
      repaint(self)
    else
      errorBox("WARNING!", "Object " + tmpObj.name +
```

```
                " already exists.");
            tmpObj := nil
        endif
    endif;
}               !!


Def  initMenuID(self)
{
menuID :=%Dictionary  (1->#saveObj
        2->#defineObj
        3->#attrDef
        4->#expandObj
        5->#deleteObj
        6->#changeMode
        7->#help
        8->#closeDD )


}    !!

/* Create DML Window to view database */
Def  changeMode(self)
{
 dMWin := new(DMWindow, self, "GladDmlMenu",
        "G L A D  D M L",self.locRect);
 dMWin.dbSchema := dbSchema;
 dMWin.selObj := selObj;
 show(dMWin,1)
}       !!

/* Deletes the selected object */
Def  deleteObj(self)
{
  changed := true;
  if selObj
    if new(ErrorBox, self, "Delete Object: " + selObj.name,
        "WARNING!", MB_YESNO) == IDYES
      remove(dbSchema, find(dbSchema, selObj));
      avail(colorTable, selObj.color);
      selObj := nil;
      repaint(self)
    endif
  else
    errorBox("No Object Selected", "Select an object to
```

```
          delete")
     endif
}           !!

/* Save the database schema */
Def  saveObj(self | aFile, fileName, dBName)
{
  addNewDb(parent);
  fileName := subString(Name,0,7) + ".sch";
  aFile := new(DBSchemaFile);
  setName(aFile, fileName);
  if exists(File, fileName, 1)
    open(aFile,1) /*write-only*/
  else
    create(aFile)
  endif;
  saveDbSchema(aFile, dbSchema);
  write(aFile, "@@" + CR_LF);
  changed := nil;
  close(aFile)
}           !!

/* Creates a new GLAD object or calls nameObj(self) for
   modification of an existing object. Performs integrity
   checking*/
Def defineObj(self)
{
  changed := true;
  nest := ATOMIC;
  tmpObj := new(GladObj);
  if selObj
    nameObj(self)
  else
    newObj := new(ObjectDialog, "", nest);
    if runModal(newObj, DEFOBJ, self) == IDOK
      defineNewObject(tmpObj, getText(newObj),
              getNestLevel(newObj));
    if isNameDefined(tmpObj, dbSchema) == nil
      add(dbSchema, tmpObj);
      repaint(self)
    else
      errorBox("WARNING!", "Object " + tmpObj.name +
           " already exists.");
```

67

```
        tmpObj := nil
      endif
        endif
  endif;
}                   !!
```

7. GladObj Class (Modified)

```
 /* for storing Glad objects. */
inherit(Object, #GladObj, #(name
pt   /*origin point of the box */
color /*to fill the box when selected*/
nesting /*true if it is a generalized object*/
attributes /*collection of name, class, and type(U or S*/
hDC  /* Display context for new object */
refCnt /*reference count*/
refNum /* reference number */
thickBorder /*true if most recently selected object*/
memberFile /*contains tuple*/
aLMWin   /*its listMemWindow*/
aOMWin   /*its oneMemWindow*/
aDscrbWin /*its describe Window*/
aNDMWin   /*its nestedDMWindow*/
), 2, nil)!!

now(GladObjClass)!!

now(GladObj)!!

/* Checks to see if object name is already defined.  Return
   'true' if the name is defined.  Otherwise returns nil. */

Def  isNameDefined(self, aColl)
{ do(aColl,
  {using(chkObj)
   if chkObj.name = name
     ^true;
   endif;});
  ^nil
}   !!
```

68

```
/* Displays newly created object */
Def  defineNewObject(self, newName, newNest)
{
  name := newName;
  if newNest == NESTED
    nesting := new(OrderedCollection, 10)
  endif;
  pt := asPoint("400@250");
  refCnt := 0;
  color := WHITE_COLOR;
}         !!


/*returns an inner box for a generalized object*/
Def nestedRect(self | aRect)
{
  aRect := new(Rect);
  aRect := rect(self);
  ^inflate(aRect,-10,-10)
}  !!


/*set the new origin point for the object rectangle
  from the current mouse position plus offset
 .Since pt is first initialized to integer,
  make sure it only gets integer value */
Def setNewOriginPt(self, mousePos, offset)
{
  pt.x := asInt(mousePos.x - offset.x);
  pt.y := asInt(mousePos.y - offset.y)
}   !!


/*erase the region a little larger than object
  rectangle in case it is displayed with a bolded
  border*/
Def eraseRect(self,hdc | hBrush)
{
  hBrush := Call CreateSolidBrush(WHITE_COLOR);
  Call FillRect(hdc,inflate(rect(self),5,5),hBrush);
  Call DeleteObject(hBrush)
}   !!


/*check whether the point is contained in the rect*/
Def containedIn(self,point | aBox)
{
```

```
    aBox := new(Rect);
    aBox := rect(self);
    if (left(aBox) <= point.x and point.x <= right(aBox)
        and top(aBox) <= point.y and point.y <= bottom(aBox))
      ^true
    else
      ^nil
    endif
} !!

/*computes the difference between the cursor point
  contained in the rectangle and the origin point
  of the object rectangle */
Def getOffset(self, point I tempPt)
{
  tempPt := new(Point);
  tempPt.x := point.x - pt.x;
  tempPt.y := point.y - pt.y;
  ^tempPt
} !!

/*returns the default rectanlge dimension for an object*/
Def rect(self I aRect)
{
  aRect := new(Rect);
  init(aRect,pt.x,pt.y,pt.x+140,pt.y+65);
  ^aRect
}    !!


/* draws an object on the window using the hdc display
   context */ Def display(self,hdc I aRect, hBrush, hPen,
   hOldBrush, hOldPen)
{
  eraseRect(self,hdc); /*first erase it*/
  /*select the color brush for filling
    used with Rectangle (via draw) */
  hBrush := Call CreateSolidBrush(color);
  /*set bkcolor for shading with DrawText*/
  Call SetBkColor(hdc,WHITE_COLOR);
  hOldBrush := Call SelectObject(hdc,hBrush);
  aRect := rect(self);
  if thickBorder
```

```
    hPen  := Call CreatePen(0,5,Call GetTextColor(hdc));
    hOldPen:= Call SelectObject(hdc,hPen);
    draw(aRect,hdc);
    Call SelectObject(hdc,hOldPen);/*restore the dc*/
    Call DeleteObject(hPen)
  else
    draw(aRect,hdc) /*with a reg. border*/
  endif;
  if nesting /*draw the inner box*/
    draw(nestedRect(self),hdc)
  endif;

  Call DrawText(hdc,lP(name),-1,aRect,
        DT_CENTER bitOr DT_VCENTER
        bitOr DT_SINGLELINE);
  Call SelectObject(hdc,hOldBrush);
  Call DeleteObject(hBrush);
  freeHandle(name)
}      `!!
```

8. GladWindow Class (Modified)

```
/* display window for GLAD */!!

inherit(Window, #GladWindow, #(dbList /* DBDialog */
dDWin /*window for data definition*/
dMWin /*window for data manipulation*/
), 2, nil)!!

now(GladWindowClass)!!

now(GladWindow)!!

/* Adds new db name to dbs list */
Def addNewDb(self)
{
  addDb(dbList, Name);
} !!

/* Checks to see if new database name is already defined.
   Returns true if it is defined, otherwise nil. */
```

71

```
Def  isNameDefined(self, aColl)
{ do(aColl,
  {using(chkName)
   if subString(chkName, 0, 7) = subString(Name, 0, 7)
     ^true;
   endif;});
  ^nil
  }    !!

Def modifyDb(self )
{
 if not(dbList)  /*not opened yet*/
  dbList := new(DBDialog)
 endif;

 dbList.state := OPEN_DB;
 if runModal(dbList,OPNDBLIST,self) == OPEN_DB

  Name := getSelDb(dbList);
  dDWin := new(DDWindow, self, "GladDdlMenu",
         "G L A D   D D L  ** " + Name,
         self.locRect);
  loadSchema(dDWin, fileNameOfSelDb(dbList));
  expanded := 0;
  show(dDWin,1)
 endif
}           !!

/*create it as tiled window; need for stand-alone appl*/
Def create(self,par,wName,rect,style)
{
 ^create(self:Window,nil,wName,rect,WS_TILEDWINDOW)
} !!

Def removeDb(self)
{
 if not(dbList)  /* not opened yet */
  dbList := new(DBDialog)
 endif;

 dbList.state := REMOVE_DB;
 runModal(dbList,RMVDBLIST,self)
}
```

```
    !!

Def openDb(self )
{
 if not(dbList)  /*not opened yet*/
  dbList := new(DBDialog)
 endif;

 dbList.state := OPEN_DB;
 if runModal(dbList,OPNDBLIST,self) == OPEN_DB
  dMWin := new(DMWindow, self, "GladDmlMenu",
          "G L A D  D M L",self.locRect);
  loadSchema(dMWin, fileNameOfSelDb(dbList));
  show(dMWin,1)
 endif
}         !!

Def makeNewDb(self)
{
 if not (dbList)
  dbList := new(DBDialog)
 endif;

 dBName := new(InputDialog, "Create Database",
         "Enter new Database name ", "");

 if runModal(dBName, INPUT_BOX, ThePort) == IDOK

  Name := getText(dBName);
  if isNameDefined(self, dbList.dbNames) == true
     /* Database name already exists */
    errorBox("WARNING!", "Database '" + Name + "' already
        exists!")
  else    /*  Database name does not exist */
   dDWin := new(DDWindow,self,"GladDdlMenu",
        "G L A D  D D L ** " + Name,
        self.locRect);
   initSchema(dDWin);
   expanded := 0;
   show(dDWin,1)
  endif
 endif;
```

```
}               !!

Def command(self, wP, lP)
{
 select
  case lP <> 0
   is ^0
  endCase

  case wP == MAKE_NEWDB
   is makeNewDb(self)
  endCase

  case wP == MODIFY_DB
   is modifyDb(self)
  endCase

  case wP == OPEN_DB
   is openDb(self)
  endCase

  case wP == REMOVE_DB
   is removeDb(self)
  endCase

  case wP == TOPHELP
   is help(self)
  endCase

  case wP == QUIT_GLAD
   is
     if dbList /*something is loaded*/
      updateDbsFile(dbList)
     endif;
     close(self)
  endCase

 endSelect;
 ^0
}               !!
```

9. NestDDWindow Class

/* A nested DD window for displaying nested objects. */!!

```
inherit(DDWindow, #NestDDWindow, #(shadeColor/*for cRect
  border*/
genObj /*generalized object of this window's nested
  objects*/
), 2, nil)!!


now(NestDDWindowClass)!!


now(NestDDWindow)!!


/* Destroys nested window */
Def WM_DESTROY(self,wp,lp)
{
  genObj.aNDMWin := nil;
  do(dbSchema,{using(obj)
          if obj.color
            avail(colorTable,obj.color);
            obj.color := WHITE_COLOR
          endif })
} !!


Def start(self,obj,aColorTbl | nullStr)
{
  genObj   := obj;
  dbSchema := obj.nesting;
  shadeColor := obj.color;
  colorTable := aColorTbl;
  nullStr := "";
  changeMenu(self,CONNECT_OBJ,lP(nullStr),0,MF_DELETE);
  freeHandle(nullStr);
  drawMenu(self);
  show(self,1)
}   !!


/*shade the outer region */
Def shadeOuterRegion(self,hdc | aRect, wd, ht, hBrush)
{
  aRect := clientRect(self);
  wd   := width(aRect);
```

```
  ht   := height(aRect);
  hBrush:= Call CreateSolidBrush(shadeColor);
  init(aRect,0,0,SHADE_BOR_WD,ht);
  Call FillRect(hdc,aRect,hBrush);
  init(aRect,0,0,wd,SHADE_BOR_HT);
  Call FillRect(hdc,aRect,hBrush);
  init(aRect,0,ht-SHADE_BOR_HT,wd,ht);
  Call FillRect(hdc,aRect,hBrush);
  init(aRect,wd-SHADE_BOR_WD,0,wd,ht);
  Call FillRect(hdc,aRect,hBrush);
  Call DeleteObject(hBrush)
}    !!

Def paint(self,hdc)
{
  shadeOuterRegion(self,hdc);
  paint(self:DBWindow,hdc)
}   !!
```

10.   ObjectDialog Class

/* Creates and controls object definition dialog box. */!!

```
inherit(Dialog, #ObjectDialog, #(dText /* Object name */
nestLevel /* Nesting Level of Object */
), 2, nil)!!
```

now(ObjectDialogClass)!!

```
/* Create an object definition dialog object. The dialog's
   instance variables are all specified as arguments.  */
Def new(self, txt, nesting | theDlg)
{ theDlg := new(self:Behavior);
  theDlg.dText := txt;
  theDlg.nestLevel := nesting;
  ^theDlg;
}  !!
```

now(ObjectDialog)!!

/* Returns the radio button selected in the new object

```
   dialog box */
Def  getNestLevel(self)
{
  ^nestLevel
} !!

/* Initialize the dialog text and caption. */
Def  initDialog(self, wp, lp)
{
   setItemText(self, FILE_EDIT, dText);
   Call CheckRadioButton(hWnd, ATOMIC, NESTED, nestLevel);
}       !!

/* Return the text that was typed in the edit box.  */
Def  getText(self)
{  ^leftJustify(dText)
}  !!

/* Handle dialog events. */
Def command(self, wp, lp)
{
 select
  case wp == ATOMIC or wp == NESTED
   is Call CheckRadioButton(hWnd, ATOMIC, NESTED, wp);
     nestLevel := wp;
  endCase

  case  wp == IDOK
   is dText := getItemText(self, FILE_EDIT);
     end(self, IDOK);
  endCase

  case wp == IDCANCEL
   is end(self, 0);
  endCase

 endSelect;
 ^0;
}   !!
```

11.   TypeDialog Class

```
/* Displays the possible attribute types which can be used
   for each given object.  */!!

inherit(Dialog, #TypeDialog, #(typeList /* List of valid
  types */
selType /* Index of the selected type */
), 2, nil)!!

now(TypeDialogClass)!!

/* Creates new instance of the TypeDialog Class */
Def  new(self, list | theDlg)
{
  theDlg := new(self:Behavior);
  theDlg.typeList := list;
  ^theDlg
}    !!

now(TypeDialog)!!

/* Selects attribute in list box and initializes the
   'Attribute Type' edit box */
Def  selItem(self)
{
  selType := Call SendDlgItemMessage (hWnd,TYPE_LIST,
        LB_GETCURSEL,0,0);
  if (selType >= 0 and selType < size(typeList))
    setItemText(parent, OBJ_ATTRIB, typeList[selType][0])
  endif;

}      !!

Def  command(self, wP, lP)
{
  select
    case wP == IDCANCEL
      is end(self, 0)
    endCase

    case (wP == ATTR_OK or (wP = TYPE_LIST and high(lP)
        = 2))
```

```
            is selItem(self);
              end(self, 0)
          endCase

        endSelect

    }                   !!


/* Inserts a new attribute type into the type list dialog
    box */
Def insertString(self,aStr | ans)
{
  ans := Call SendDlgItemMessage(hWnd, TYPE_LIST,
          LB_INSERTSTRING,-1,lP(aStr));
  freeHandle(aStr);
  ^ans
}      !!


/* Initialize the listbox;  the method is the Actor
    equivalent of WM_INITDIALOG */
Def  initDialog(self, wP, lP)
{
  do(typeList,
    {using(elem) insertString(self, subString(elem[0]
                + "        ", 0,10) + ": "
                + elem[1])});
}
    !!
```

LIST OF REFERENCES

1. U.S. Congress, House, *Computer Security Act of 1987*, H. R. 100-153, 100th Cong., 1st sess., 1987.

2. Korth, H. F. and Silberschatz, A., *Database System Concept*, McGraw-Hill Book Company, 1986.

3. Wu, C. T., *GLAD: Graphics Language for Database*, Naval Postgraduate School Report NPS52-87-030, Monterey, California, 1987.

4. Microsoft Corporation, *Microsoft Windows User's Guide*, Microsoft Press, 1987.

5. MacLennan, B. J., *Principles of Programming Languages*, Holt, Rinehart, and Winston, 1987.

6. Norman, D.A., and Draper, S. W., ed., *User Centered System Design*, Lawrence Erlbaum Associates, 1986.

7. Thomas, J. C., and Schneider, M. L., ed., *Human Factors in Computer Systems*, Ablex Publishing Corp., 1984.

8. Duff, C. and others, *Actor Language Manual*, Evanston, Illinois: The Whitewater Group, 1987.

9. Kaehler, T., and Patterson, D., "A Small Taste of Smalltalk," *BYTE*, August, 1986.

BIBLIOGRAPHY

Cox, B., and Hunt, B., "Objects, Icons, and Software-ICS," *BYTE*, August 1986.

Davis, M., "Smalltalk/V Release 1.2," *BYTE*, June 1987.

Duff, C. B., "Designing an Efficient Language," *BYTE*, August 1986.

Monk, A.w, ed., *Fundamentals of Human-Computer Interaction*, Academic Press, 1984.

Moskowitz, L., "Actor 1.0," *BYTE*, September 1987.

Pascoe, G. A., "Elements of Object-Oriented Programming," *BYTE*, August 1986.

Petzold, C., *Programming Windows*, Microsoft Press, 1988.

Sanders, M. S., and McCormick, Ernest J., *Human Factors in Engineering and Design*, McGraw-Hill Book Company, 1987.

Tesler, L., "Programming Experiences," *BYTE*, August 1986.

INITIAL DISTRIBUTION LIST